# Dynamic Language Runtime

**Bill Chiles and Alex Turner**

***Reading this Document:***

> ***The first section conveys mission, goals, and value propositions for the DLR.***
>
> ***The second section is a bird's eye view of the architecture with a few sentences about each DLR component.***
>
> ***The following sections drill into the major areas and components with conceptual introductions in moderate detail. These sections lift some text from more detailed documents that provide further reading such as samples, API references, etc. There are some sub sections here that are not found in the other documents.***


1 Introduction ..... 3

1.1 Key DLR Advantages ..... 3

1.2 Open Source Projects ..... 4

1.3 Why Dynamic Languages ..... 4

2 Architecture Introduction ..... 6

2.1 Common Hosting ..... 7

2.2 Runtime ..... 8

2.3 Language Implementation ..... 9

3 Common Hosting Model ..... 9

3.1.1 Architecture Introduction ..... 10

3.1.2 Level One -- Script Runtimes, Scopes, and Executing Files and Snippets ..... 12

3.1.3 Level Two -- Engines, Compiled Code, Sources, and Object Operations ..... 13

3.1.4 Level Three -- Full Control, Remoting, Tool Support, and More ..... 14

4 Runtime ..... 17

4.1 Dynamic Call Sites ..... 17

4.1.1 Before Dynamic Call Sites ..... 17

4.1.2 Language Interoperability ..... 18

4.1.3 Creating Dynamic Call Sites ..... 19

4.2 Rules ..... 19

4.3 Binders and CallSiteBinder ..... 20
4.4 CallSite<T> and Caching ..... 21
4.4.1 L0 Cache: CallSite's Target Delegate ..... 21
4.4.2 L1 Cache: CallSite's Rule Set ..... 22
4.4.3 Other Optimizations ..... 22
4.5 DynamicObject and ExpandoObject ..... 22
4.5.1 DynamicObject ..... 23
4.5.2 ExpandoObject ..... 24
4.6 Default Binder -- Runtime Utility for Language Implementers ..... 24
5 Language Implementation ..... 25
5.1 Expression Trees ..... 26
5.1.1 Expression-based Model ..... 26
5.1.2 Reducible Nodes ..... 27
5.1.3 Bound, Unbound, and Dynamic Nodes ..... 28
5.1.4 Iteration, Goto's, and Exits ..... 30
5.1.5 Assignments and L-values ..... 32
5.1.6 Blocks, Scopes, Variables, ParameterExpression, and Explicit Lifting ..... 34
5.1.7 Lambdas, Exits, and Result Types ..... 35
5.1.8 Generators (Codeplex only) ..... 36
5.1.9 Serializability ..... 37
5.1.10 Shared Visitor Support ..... 37
5.1.11 Annotations (Debug/Source Location Only) ..... 38
5.2 LanguageContexts ..... 38
5.3 Type System ..... 39
5.3.1 Problems with Wrappers ..... 39
5.3.2 Object and IDynamicMetaObjectProvider ..... 40
5.4 IDynamicMetaObjectProvider and DynamicMetaObject ..... 40
5.4.1 IDynamicMetaObjectProvider ..... 41
5.4.2 DynamicMetaObject ..... 41
5.4.3 DynamicMetaObjectBinder ..... 42

# 1 Introduction

The Dynamic Language Runtime (DLR) is a set of libraries built on the CLR to support dynamic language implementations on .NET. The DLR's mission is to enable an ecosystem of dynamic languages on .NET. A key value proposition of the .NET CLR is supporting multiple languages and allowing them to interoperate with each other. Dynamic languages have become very popular in the last several years. Customers want to use their favorite dynamic language and have great .NET interoperability for building applications and providing scripting for applications. The DLR makes it very easy to develop dynamic languages on .NET.

The DLR also has support for existing languages on .NET. If you already have a language implemented on .NET, you might want to add dynamic dispatch capabilities. As with C# 4.0, this enables the language to support very nice expressions (syntactically light) when working with dynamic objects via COM, HTML DOM, or .NET reflection. The DLR provides an entry point for you to just use the fast dynamic dispatch.

The DLR has high-level support for library authors too. If you have a library for crawling through XML or working with JSON objects, you'd really like to enable your objects to appear as dynamic objects to C# 4.0 and dynamic languages. This lets consumers write syntactically simpler and more natural code for accessing members, drilling into, and operating on your objects. The DLR provides two high-level helper objects so that library authors do not have to work at the level of language implementers to support dynamic operations.

The DLR provides three key components:

- language implementation services with language interoperability model
- dynamic language runtime services with fast dynamic dispatch and library support
- common hosting APIs across languages

Language implementers get great .NET interoperability. They also have several mechanisms for keeping their language true to its semantics and feel. See sections 1.1 and 2 for summaries of these components.

The key goals of the DLR are making it easy to

- port dynamic languages to .NET
- add dynamic features to your existing language
- author libraries whose objects support dynamic operations
- employ dynamic languages in your applications and frameworks.

The following sections provide an overview of the DLR, its overall architecture, and introductions to its key components.

## 1.1 Key DLR Advantages

**For language implementers the DLR lowers the bar considerably for porting a language to .NET**. Traditionally, implementers needed to build lexers, parsers, semantic analysis, optimization passes, code generation, runtime support, and so on. Virtual machines lowered the bar so that languages could emit a higher-level intermediate language instead of fully optimized machine code. The DLR essentially only requires languages to produce a bound abstract semantic tree (.NET Expression trees) and some runtime helpers if needed. The DLR and .NET do the rest of the work.

**Languages implemented using the DLR continually benefit from improvements lower down the stack**. Microsoft designed the .NET Framework to support a broad range of programming languages on the Common Language Runtime (CLR). The CLR provides shared services to these languages including garbage collection, just-in-time (JIT) compilation, a sandboxed security model, and support for tools integration. Sharing libraries and frameworks allows languages new to the CLR to build on the work of others. When .NET releases a new version with performance gains, for example, your language immediately benefits. When the DLR adds optimizations such as better compilation, language performance improves for everyone.

**In the Dynamic Language Runtime we provide common language interoperability, fast dynamic invocation, and some utilities**. The language interoperability story is based on a protocol for objects implemented in one language to be used by other languages. With dynamic typing, the object is king for determining if it can support a particular message or operation sent to it. Similarly the DLR enables dynamic objects to participate in a message passing protocol for negotiating how to perform abstract operations on any object. The fast dynamic dispatch is based on polymorphic inline caching. Dynamic objects can also participate in the fast dynamic invocation so that a particular call site can cache implementations of abstract operations from the calling language or from objects implemented by other languages.

**With common hosting APIs, applications can use any language supporting the DLR hosting model**. At a very high level, the DLR provides multiple script runtime environments per AppDomain, as well as remote script runtimes in other AppDomains. Hosts can inject global variable bindings and execute files or snippets of code in the context of those bindings. Hosts can create individual scopes of variable bindings and execute code in them. After executing code, hosts can extract scope variables or globals to hook up event handlers, command implementations, etc. Hosts can also invoke dynamic operations on dynamic objects living in the script runtimes.

## 1.2 Open Source Projects

Most of our DLR languages and all the DLR source code are available on CodePlex and RubyForge. The sources are available under the Microsoft Public License, which is Open Software Initiative approved. The DLR and all of our open source languages are available for one-stop shopping at www.codeplex.com/dlr, where we will add more samples, specs, and getting started documentation over time. Microsoft currently provides two DLR languages, IronPython and IronRuby. IronPython is available open source at www.codeplex.com/ironpython. IronRuby is available open source at http://rubyforge.org/projects/ironruby.

## 1.3 Why Dynamic Languages

This section introduces motivations for supporting Dynamic languages or adding dynamic features to a static language, which is occurring more and more these days. This debate with language designers is a classic religious battle. This section is not in any way a complete rhetoric for why you should embrace dynamic languages, and in no way does it try to say you should only use dynamic languages. This is just a brief treatment of some common reasons people cite for interest in dynamic languages.

Dynamic languages are one of those cyclic technologies that have become vogue again. They were hot in the 80s for scripting and in any startup claiming to do AI. They became hot again due to the web. The web is essentially built on dynamic languages and View Source. Due to the web, dynamic language are not only here to stay, but static languages are adopting dynamic features to make them more productive for web development and working with inherently dynamic modeling objects.

Many dynamic languages are popping up over the last 10+ years: JavaScript, PHP, Ruby, Python, ColdFusion, LUA, Cobra, Groovy, Newspeak, and more. Some popular dynamic languages (or those with dynamic features) have been around for quite a while: Perl, VB, Smalltalk, Lisp, and Scheme. While some of these aren't used as prevalently today as they were at one time, they are still popular with some programmers and in use today.

The communities around dynamic languages are very strong. They have very deep passions for their languages because they feel their languages lend themselves to a high degree of productivity. These programmers will use dynamic languages whenever possible. They will use them for infrastructure that maintains systems, scripting of applications, building whole applications, and so on.

A key productivity aspect of many dynamic languages is the ability to use a rapid feedback loop (REPL, or read-eval-print loop). This lets you enter snippets of code and hit enter to immediately see the results of executing the code. The ability to iteratively develop code by working with live objects is a very powerful mechanism for discovering how to use an API and experimenting with solutions to problems. Because dynamic languages are tolerant of unimplemented surface area, REPLs also support simultaneous top-down and bottom-up development. This means you can start with high-level functions and make calls to as-yet unimplemented functions. Then you can fill in underlying implementation or low-level utilities as you need them or further develop a branch of code.

Dynamic languages lend themselves to refactoring and making code changes more rapidly. Code is always evolving as implementation feeds back on design or as requirements change. You do not have to fix up static type declarations everywhere to make logical changes in your application. There are often a lot fewer textual changes to the code to update the logic. Dynamic languages also usually support features such as optional or named parameters that aid making changes to definitions and having a lighter-weight experience fixing up call sites.

Meta-programming with macros or syntactic flexibility are features often provided by dynamic languages. These are powerful mechanisms for defining domain-specific languages or extending a language in a more natural way than adding a library of functions. Macros can make code much more expressible, increasing productivity, in mature systems with domain-specific patterns or coding needs.

Dynamic languages make great glue code for snapping together applications or extensions from a palette of components. Due to the productivity of dynamic languages, ability to re-load code and keep running in a live runtime, and ability to work with types loosely, they make excellent scripting languages. Applications can host dynamic languages, provide an object model, and easily let customers extend the application with new commands and functionality.

Some common uses of dynamic languages:

- Scripting applications
- Building web sites

- Test harnesses
- Server farm maintenance
- One-off utilities or data crunching

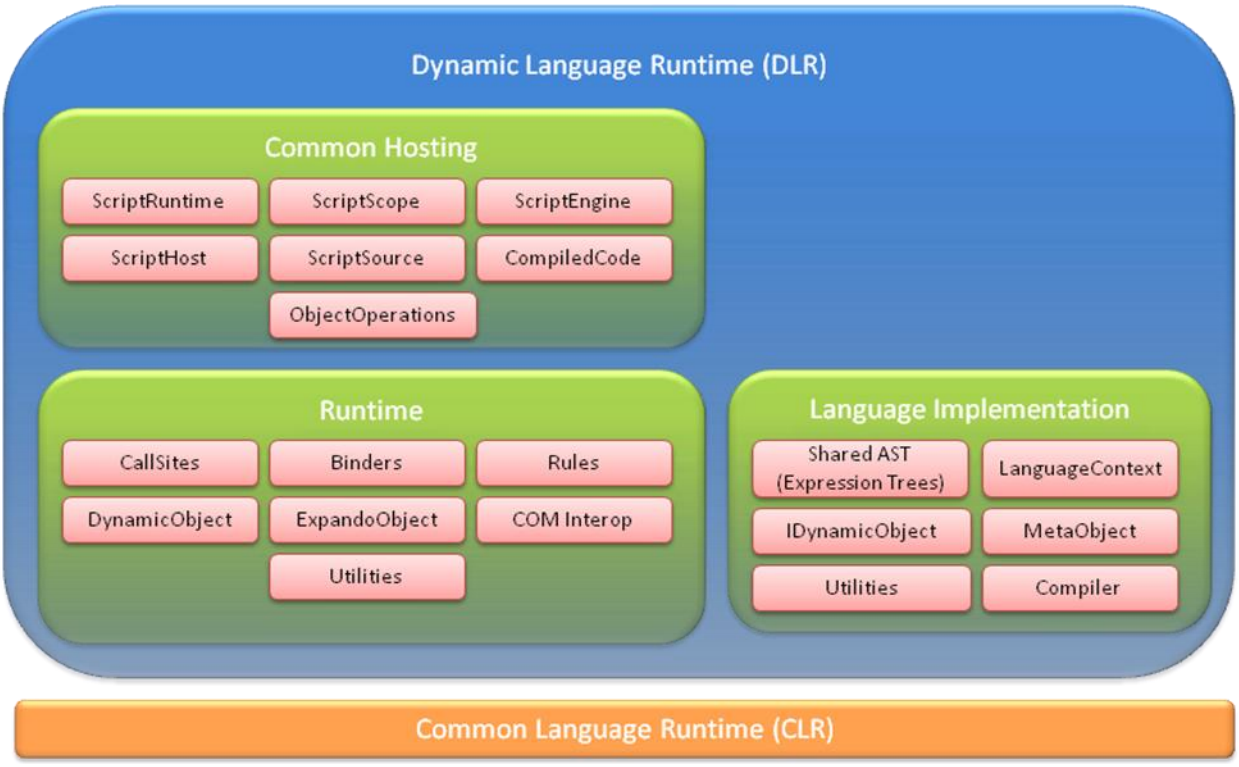
## 2 Architecture Introduction

Microsoft is building the Dynamic Language Runtime (DLR), which adds to the CLR a set of services designed specifically for the needs of dynamic languages. The CLR has good support for dynamic languages already, and IronPython-1.0 is a good example (www.codeplex.com/ironpython). The DLR adds functionality to make it easier to port and use dynamic languages on the CLR. The DLR key pieces are:

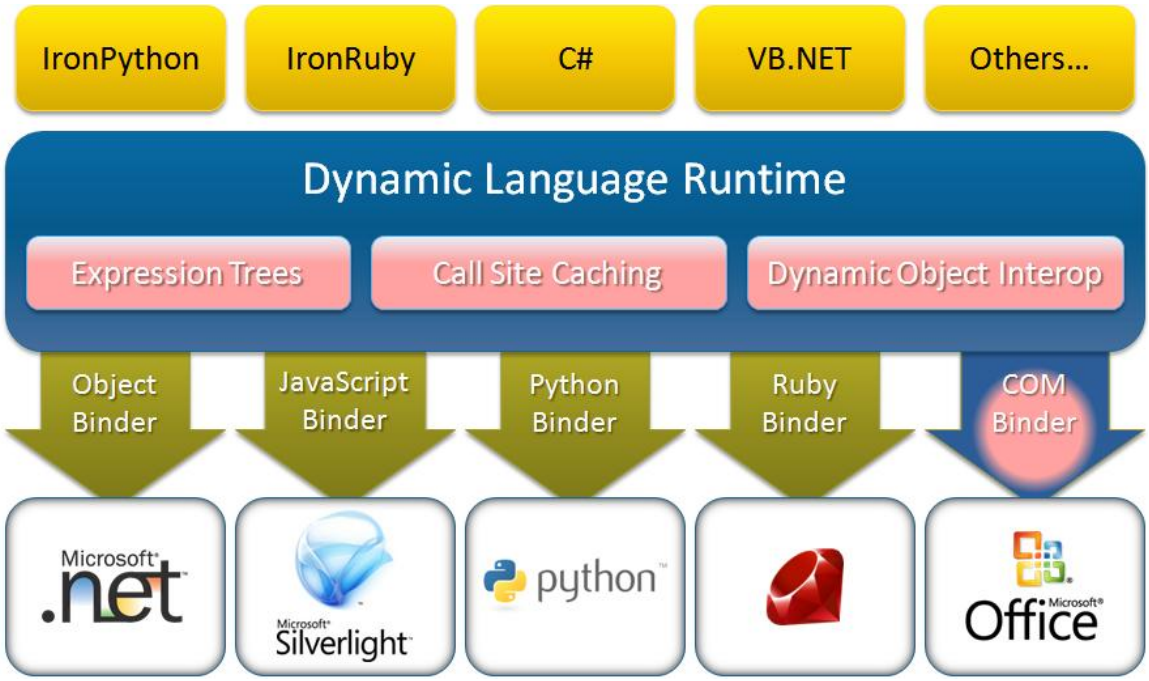
- common hosting model
- shared abstract semantic tree representation (Expression Trees)
- support to make it easy to generate fast dynamic code (DynamicSites, SiteBinders, Rules).
- shared dynamic type system (IDynamicMetaObjectProvider)
- utilities (default binder, tuples, big integers, adapters for existing static types to participate in the IDynamicMetaObjectProvider protocol, support for new static types to easily participate in IDynamicMetaObjectProvider protocol, etc.)

With these additional features, it is much easier to build high-quality dynamic language implementations for .NET. Also, these features enable dynamic languages built on the DLR to share libraries written in other dynamic languages or static languages on the .NET CLR.

Let's take a look at 1000 words using picture compression. Here is a conceptual architecture diagram (not all labels are actual type names), followed descriptions of the boxes:



In CLR 4.0, the parts of the DLR shipping are depicted here (pink boxes), squeezing some boxes together:




The following sub sections contain a very brief overview of the architectural pieces depicted above.

## 2.1 Common Hosting

ScriptRuntime -- This is the starting point for hosting. The ScriptRuntime represents global script state, such as referenced assemblies and a global object (a ScriptScope). You can have more than one ScriptRuntime per .Net AppDomain. You can load assemblies from which a ScriptRuntime coordinates with ScriptEngines to make namespaces and types available to script code.

ScriptScope -- This essentially represents a namespace. Hosts can bind variable names in ScriptScopes, fetch variable values, etc. Hosts can execute code within different scopes to isolate free variable resolutions.

ScriptEngine -- These are the work horse and represent a language's semantics. They offer various ways to execute code and create ScriptScopes and ScriptSources. You can have a single instance of a given language's engine in a ScriptRuntime instance.

ScriptSource -- These offer methods for reading and executing code in various ways from different kinds of sources.

CompiledCode -- These represent code that has been compiled to execute repeatedly without having to compile it each time, and they have a default ScriptScope the code runs in. The

default scope may have optimized variable storage and lookup for the code. You can always execute the code in any ScriptScope if you need it to execute in a clean scope each time, or you want to accumulate side effects from the code in another scope.

ScriptHost -- This lets you provide a custom PlatformAdaptationLayer object to override file name resolution. For example, you might only load files from a particular directory or go to a web server for files. A host communicates its sub type of ScriptHost to the DLR when it creates a ScriptRuntime. Many hosts can just use the DLR's default ScriptHost.

ObjectOperations -- These provide a large catalogue of object operations such as member access, conversions, indexing, and operations like addition. There are several introspection and tooling support services. You get ObjectOperation instances from engines, and they are bound to their engines for the semantics of the operations.

## 2.2 Runtime

CallSites -- These provide a caching mechanism per operation or call in dynamic code. Without CallSites, every time you executed "a + b" the DLR would have to fully search for how to add a and b. With CallSites the DLR can store a rule for what to do with a and b given their characteristics (which typically means their type).

Binders -- These represent a language's semantics for performing a specific operation in a CallSite, including any metadata about the CallSite. Binders get called when there is a cache miss in a CallSite, and they inspect the operands of the CallSite to compute how to perform the operation in the CallSite. Binders communicate how to perform operations to CallSites using Expression Trees.

Rules -- These represent a single computation or binding of how to perform an operation in a CallSite. Rules have a test for when the rule is valid (typically just testing the static runtime type of the operands). Rules have a target or expression for how to perform the operation, given the characteristics of the operands guaranteed by the Rule's test. The test and target expressions come from the binders.

DynamicObject -- This is a convenience type for library authors. DynamicObject has virtual methods for all the abstract operations you can perform on via IDynamicMetaObjectProvider and DynamicMetaObject. You can simply derive from DynamicObject and implement the operations you care about. You do not have to create Expression Trees or create your own DynamicMetaObject to providing binding to a CallSite. The DLR does that for you.

ExpandoObject -- This is a convenience object for library authors. They can delegate to it or use it directly as a simple dynamic object that lets you add, set, and remove members to the object at runtime.

Utilities -- There are some helper classes for implementing binders, producing rules, etc. For example, there is a DefaultBinder that IronPython and IronRuby share for doing .NET binding. Some languages specialize their .NET binding, such as C# and VB, but other languages may be able to make use of the DefaultBinder.

COM Interop -- The DLR provides an IDynamicMetaObjectProvider and DynamicMetaObject for COM objects to participate in the dynamic operations and CallSites. The DLR uses IDispatch to bind to COM objects.

### 2.3 Language Implementation

Shared AST (Expression Trees) -- This is one of the core pillars of the DLR. We have extended LINQ Expression Trees to include control flow, assignment, etc. We also ship the sources for all of Expression Trees v1 and v2 (the new stuff for the DLR). Expression Tres are part of lowering the bar for porting languages to .NET, and we use them in the binders and DynamicMetaObject protocols.

LanguageContext -- These are the objects that represent a language that is implemented on the DLR and support the DLR's hosting model. These are the workhorse for executing code, with many members that support various higher-level features in the DLR.

IDynamicMetaObjectProvider -- This is the type implemented by languages and library authors who want to control the dispatching and binding logic for the DLR's interoperability protocol. It is foremost just a jumper to get DynamicMetaObjects.

DynamicMetaObject -- This type is implemented by languages and library authors to represent an object in an abstract operation and how to perform binding for that object. A DynamicMetaObject has several methods representing the abstract operations such as fetching members, setting members, invoking a member, invoking the object itself, instantiating the object, or performing standard operations (addition, multiplication, etc.).

Compiler -- The DLR ships an Expression Tree compiler, Expression<T>.Compile that returns a delegate for invoking the code represented by the tree.

Utilities -- There are some helper classes for implementing languages. The DLR has a GeneratorExpression that can be used in a LambdaExpression for creating what C# calls iterators. The DLR rewrites the expression tree, which contains YieldExpressions, to a new tree that open codes the state machine for re-entering the iteration and re-establishing any dynamic state (for example, try-catch's). There are math helpers such as BigInteger, Complex, and Tuples. There is a GlobalVariableExpression and optimized module support for creating supporting the DLR hosting model and having faster ScriptScopes behind CompiledCode.

## 3 Common Hosting Model

One of the top DLR features is common hosting support for all languages implemented on the DLR. The primary goal is supporting .NET applications hosting the DLR's ScriptRuntime and engines for the following high-level scenarios:

- SilverLight hosting in browsers
- MerlinWeb on the server
- Interaction consoles where the ScriptRuntime is possibly isolated in another app domain.
- Editing tool with colorization, completion, and parameter tips (may only work on live objects in v1)

A quick survey of functionality includes:

- Create ScriptRuntimes locally or in remote app domains.
- Execute snippets of code.
- Execute files of code in their own execution context (ScriptScope).

- Explicitly choose language engines to use or just execute files to let the DLR find the right engine.
- Create scopes privately or publicly for executing code in.
- Create scopes, set variables in the scope to provide host object models, and publish the scopes for dynamic languages to import, require, etc.
- Create scopes, set variables to provide object models, and execute code within the scopes.
- Fetch dynamic objects and functions from scopes bound to names or execute expressions that return objects.
- Call dynamic functions as host command implementations or event handlers.
- Get reflection information for object members, parameter information, and documentation.
- Control how files are resolved when dynamic languages import other files of code.

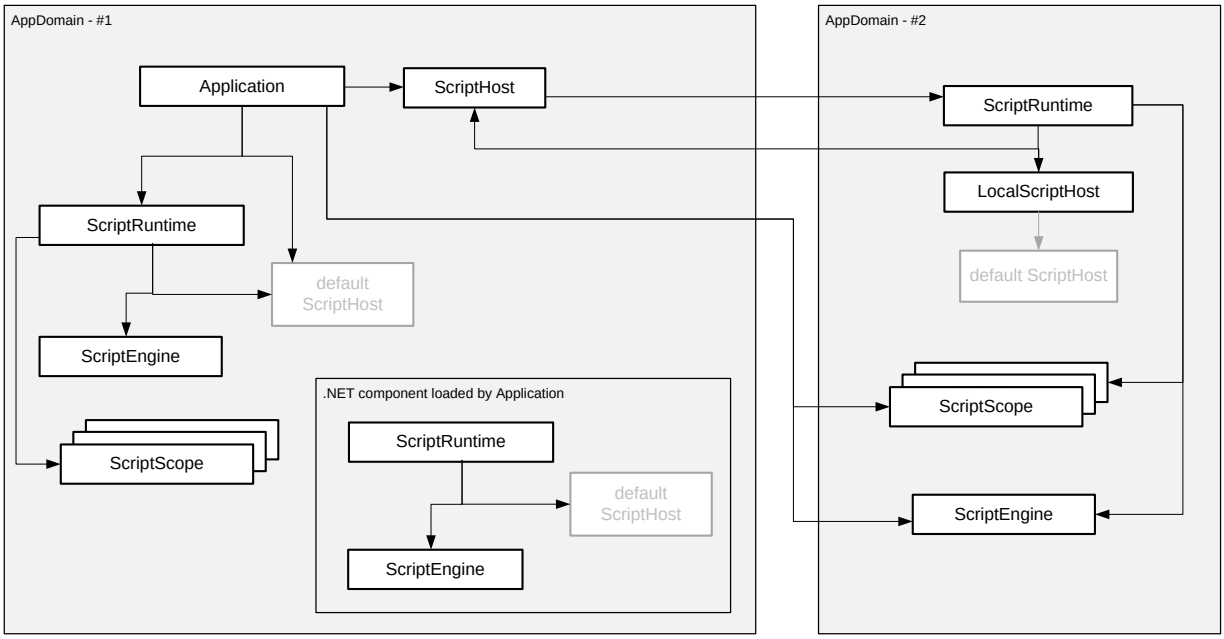
Hosts always start by calling statically on the ScriptRuntime to create a ScriptRuntime. In the simplest case, the host can set globals and execute files that access the globals. In more advanced scenarios, hosts can fully control language engines, get services from them, work with compiled code, explicitly execute code in specific scopes, interact in rich ways with dynamic objects from the ScriptRuntime, and so on.

### 3.1.1 Architecture Introduction

The hosting APIs can be grouped by levels of engagement. Level One uses a couple of types for executing code in scopes and working with variable bindings in those scopes. Level Two involves a few more types and supports more control over how code executes, using compiled code in various scopes, and using various sources of code. Level Three opens up to several advanced concepts such as overriding how filenames are resolved, providing custom source content readers, reflecting over objects for design-time tool support, providing late bound variable values from the host, and using remote ScriptRuntimes.

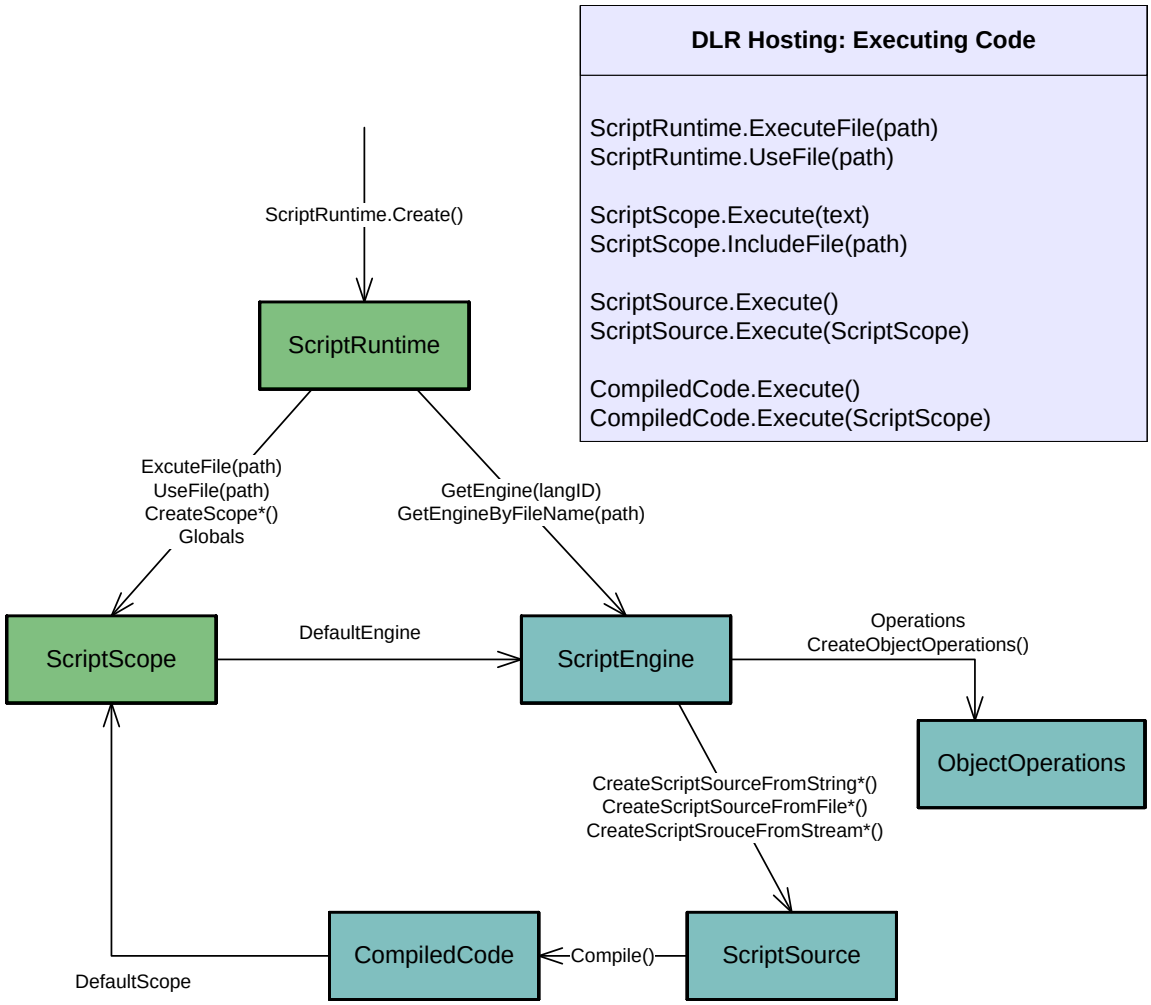
There are three basic mechanisms for partially isolating state for code executions within a process. .NET offers Appdomains, which allows for code to run at different trust levels and to be completely torn down and unloaded. The DLR offers multiple ScriptRuntimes within an AppDomain, each having its own global object of name bindings, distinct references to .NET namespaces from specified assemblies, distinct options, etc. The DLR also provides ScriptScopes which provide variable binding isolation, and you can execute code in different scopes to work with distinct bindings of free variable references.

The following diagram shows conceptually how hosts relate to ScriptRuntimes and other hosting objects:




It is important for the DLR to support distinct ScriptRuntimes within .NET's AppDomains for a couple of reasons. First, key customers betting on us require the lighter-weight isolation than what AppDomains provide. Second, consider two independent .NET components loading as extensions to an application. Each component wants to provide scripting support to end users. The components should be able to do so without having to worry about how they might clash with other script-enabled components. Multiple ScriptRuntimes also makes the main application's job easier since it does not have to provide a model for independent components getting access to and coordinating code executions around a central ScriptRuntime.

There's a rich set of ways to execute code. Our goal is to strike a balance between convenient execution methods on various objects and keeping redundancy across the types to a minimum. The diagram below shows how to navigate to the key objects for running code. The Sections on Levels One and Level Two below talk about types in this diagram (green being Level One types):



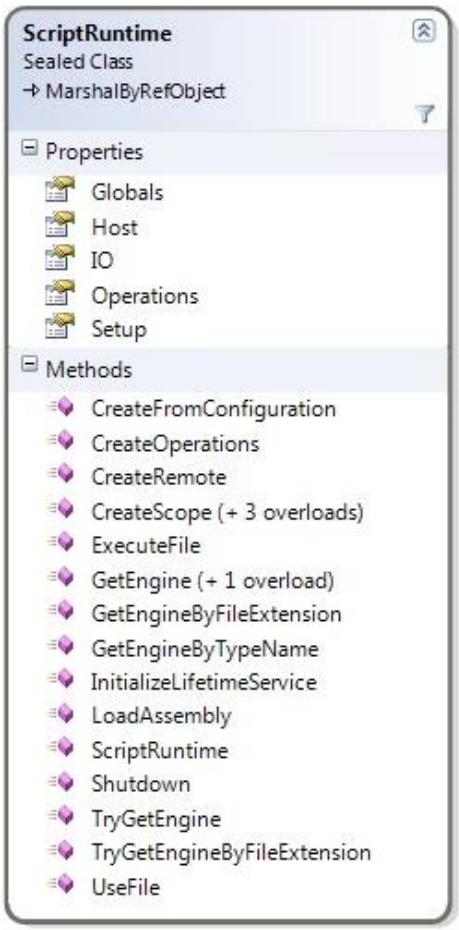

### 3.1.2 Level One -- Script Runtimes, Scopes, and Executing Files and Snippets

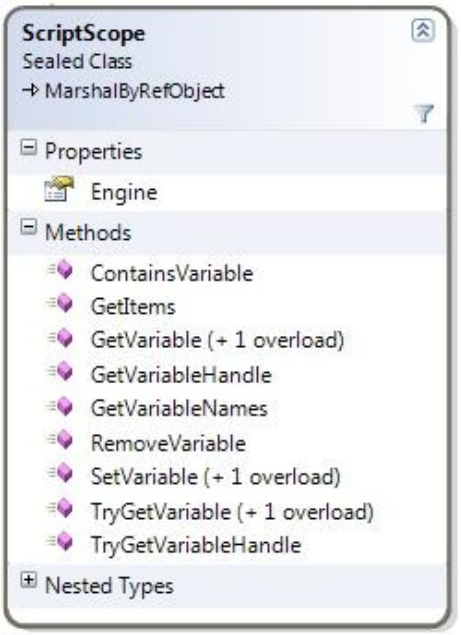
For simple application programmability, you want to provide a host object model that dynamic languages code can use. You then want to execute files of code that consume that object model. You may also want to get the values of variables from the dynamic language code to use dynamic functions as command implementations or event handlers.

There are two types you will use at this level. The ScriptRuntime class is the starting point for hosting. You create a runtime with this class. The ScriptRuntime represents global script state, such as referenced assemblies and a global object (a ScriptScope). The ScriptScope class essentially represents a namespace. Hosts can bind variable names in ScriptScopes, fetch variable values, etc. Hosts can execute code within different scopes to isolate free variable resolutions.

There are a lot of members on these types because they are also used for Level Two and Level Three. For Level One you only need a few members and can ignore the rest. You need to create a ScriptRuntime, from which you might use ExecuteFile or Globals. The Globals object lets you set variables to provide access to a host object model. From ScriptScope, you will likely only use GetVariable and SetVariable.

These types are shown in the diagram:




The ScriptRuntime.GetEngine and ScriptScope.Engine are bridges to more advanced hosting functionality. In Level Two and Level Three scenarios, the other members of ScriptRuntime and ScriptScope will be useful along with ScriptEngine.

### 3.1.3 Level Two -- Engines, Compiled Code, Sources, and Object Operations

The next level of engagement involves operating directly with engines and abstractions of source code and compiled code. You can compile code and run it in any scope or its default scope. You also have more control over how you provide sources to the DLR.

Besides the new types you'll use for Level Two scenarios, you will likely use more of the ScriptRuntime and ScriptScope classes. You'll certainly use their members to get to engines. You'll likely use more flavors of getting and setting variables on scopes as you have richer interactions with dynamic code. You might use the ability of scopes to support executing multiple languages within one scope, using execute methods on engines.

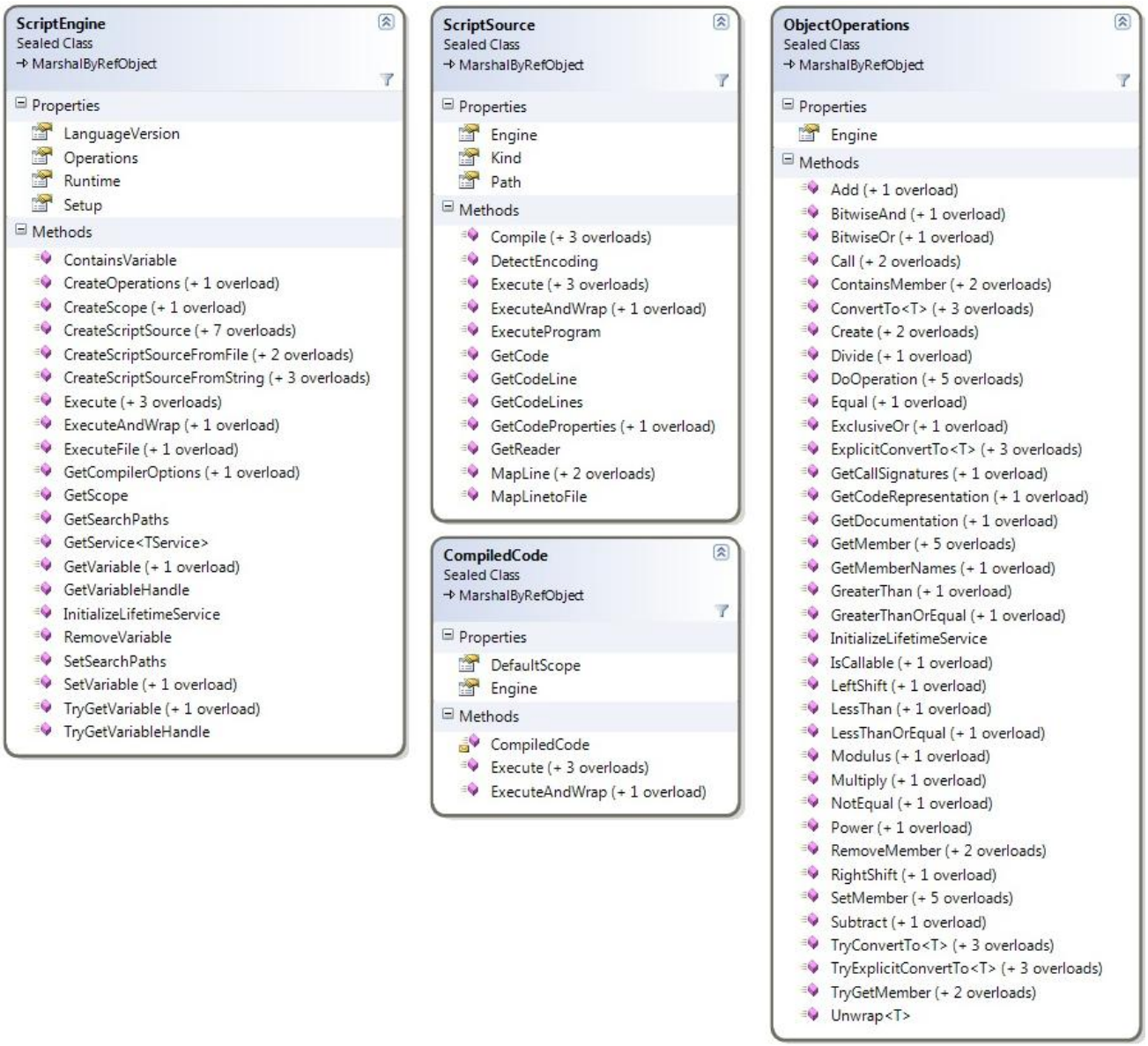
The main new types you'll use in Level Two scenarios are ScriptEngines, ScriptSources, and ObjectOperations. ScriptEngines are the work horse. They offer various ways to execute code and create ScriptScopes and ScriptSources. ScriptSources offer methods for executing code in various ways from different kinds of sources.

You may use ScriptRuntime.LoadAssembly to makes namespaces and types available to script code. The ScriptRuntime coordinates with ScriptEngines, but how script code accesses .NET

namespaces and types is language-dependent. The language may require an 'import', 'using', or 'require' construct, or the language may put first class dynamic objects in ScriptRuntime.Globals.

ObjectOperations provide a large catalogue of object operations such as member access, conversions, indexing, and operations like addition. There are several introspection and tool support services that we'll discuss in Level Three scenarios. You get ObjectOperation instances from engines, and they are bound to their engines for the semantics of the operations.

These are the main types of level two:



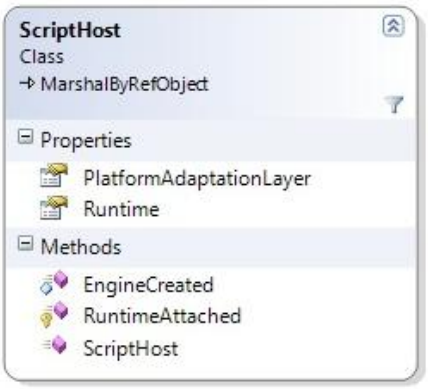

### 3.1.4 Level Three -- Full Control, Remoting, Tool Support, and More

Level three engagement as a host gives you full control over the ScriptRuntime. There are many things hosts can do at this level from controlling runtime configuration, to handling runtime

exceptions, to using remote ScriptRuntimes, to providing full programming tool support with completion, parameter info pop-ups, and colorization.

With level three support, you can create a ScriptRuntimeSetup object to control which languages are available or set options for a ScriptRuntime's behavior. For example, you can limit the ScriptRuntime certain versions of particular languages. You can also use .NET application configuration to allow users to customize what languages are available.

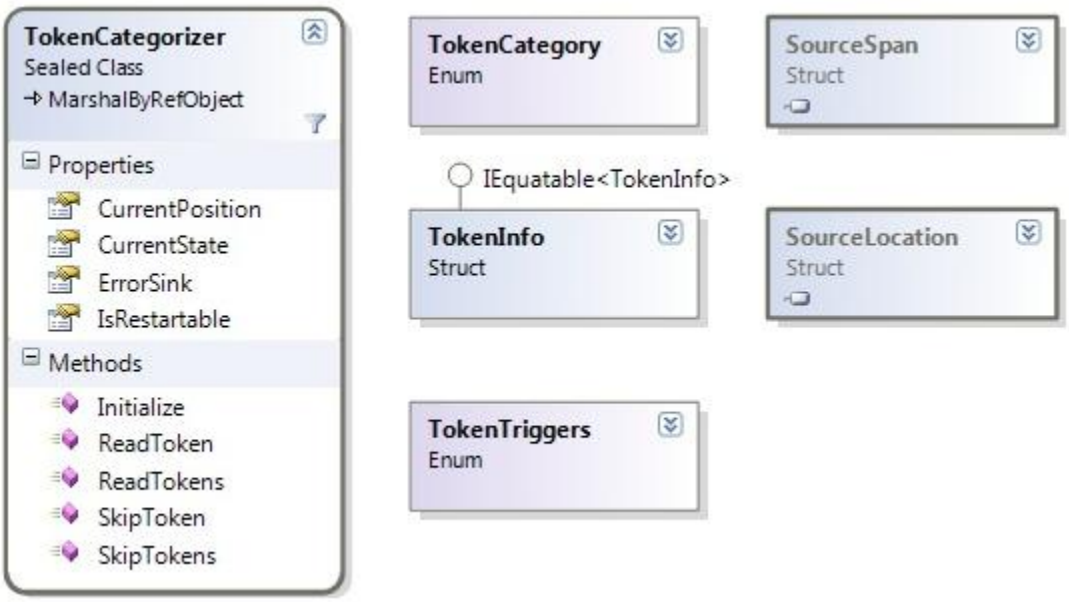
Another simple mechanism in level three is deriving from ScriptHost. This lets you provide a custom PlatformAdaptationLayer object to override file name resolution. For example, you might only load files from a particular directory or go to a web server for files. A host communicates its sub type of ScriptHost to the DLR when it creates a ScriptRuntime. Many hosts can just use the DLR's default ScriptHost. ScriptHost looks like:



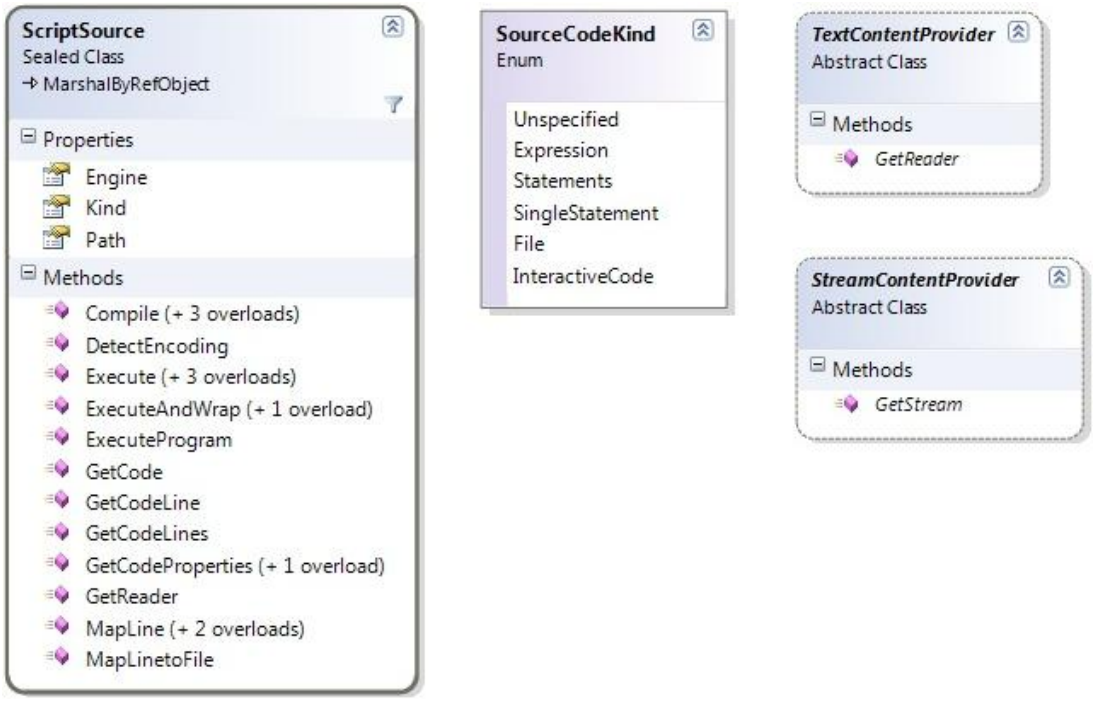

The ObjectOperations class provides language-specific operations on objects, including some tool building support. ObjectOperations includes introspection of objects via members such as GetMemberNames, IsCallable, GetCallSignatures, GetDocumentation, and GetCodeRepresentation. These give you a language-specific view in that you see members of objects and signatures from the flavor of a specific language. For example, you would see the Python meta-programming members of objects that Python manifests on objects. ObjectOperations enables you to build tool support for dynamic languages on the DLR, but you need another set of objects for parsing code.

Hosts can get parsing support for providing colorization in editors and interpreters. The following are the types used for that functionality:

*This will definitely change in the future. For now, it is a place holder and represents some prototyping we did.*




Hosts that implement tools for programmers will likely also create ScriptSources and implement TextContentProviders so that tokenizers can read input directly from the host's data structures. For example, an editor with a file in memory (in an editor buffer) could implement a TextContentProviders that reads input directly from the buffer's data structure. The types relating to sources are:




Advanced hosts can also provide late-bound values for variables that dynamic languages access. Each time the variable's value is fetched the host can compute a new value or deliver one that's

cached in its data structures. Hosts do this by being the implementer of an IDynamicMetaObjectProvider that the hosts supply when creating a ScriptScope. A simple way to implement this interface is to derive from DynamicObject which implements the interface. Then just override methods such as TryGetMember and TrySetMember. If you don't need to do fancy late binding of names, you could use ExpandoObject as a fast property bag.

# 4 Runtime

It is hard to separate language implementation concepts from the runtime concepts with dynamic languages. However, we try to do so by defining the runtime aspects of the DLR as DynamicSites, SiteBinders, and Rules for fast dynamic invocation. We include higher-level objects as helpers for library authors who want their objects to participate well in dynamic operations -- DynamicObject and ExpandoObject. We also include utilities, default binding helpers, and COM IDispatch interoperability.

## 4.1 Dynamic Call Sites

Dynamic call sites allow dynamic language code to run fast. They manage the method caching for performing specific operations on specific types of objects. The dynamic sites mechanism the DLR uses is based on research and experience with tried-and-true dynamic language implementations. For deeper discussion than provided here search the web for dynamic language method caching or polymorphic inline method caching.

Dynamic language performance is hindered by the extra checks and searches that occur at each call site. Straightforward implementations have to repeatedly search class precedence lists for members and potentially resolve overloads on method argument types each time you execute a particular line of code. In an expression such as "o.m(x, y)" or "x + y", dynamic languages need to check exactly what kind of object o is, what is m bound to for o, what type x is, what type y is, or what '+' means for the actual runtime types of x and y. In a statically typed language (or with enough type hints in the code and using type inference), you can emit exactly the instructions or runtime function calls that are appropriate at each call site. You can do this because you know from the static types what is needed at compile time.

Dynamic languages provide great productivity enhancements and powerful terse expressions due to their dynamic capabilities. However, in practice code tends to execute on the same types of objects each time. This means you can improve performance by remembering the results of method searches the first time a section of code executes. For example, with "x + y", if x and y are integers the first time that expression executes, we can remember a code sequence or exactly what runtime function performs addition given two integers. Then each time that expression executes, there is no search involved. The code just checks that x and y are integers again, and dispatches to the right code with no searching. The result can literally reduce to inlined code generation with a couple of type checks and an add instruction depending on the semantics of an operation and method caching mechanisms used.

### 4.1.1 Before Dynamic Call Sites

To step back a moment, let's look at what language implementations did before polymorphic inline caching. They would use runtime helper methods that open-coded all the checks for

common type combinations that showed up in applications for different kinds of operations. For example, in the original implementation of IronPython before the DLR, there were vast quantities of code in associated helper classes that looked like this:

```
object Add(object x, object y) {
    if (x is int) return IntOps.Add((int)x, y);
    if (x is double) return DoubleOps.Add((double)x, y);
    // ... lots more special cases ...
    // Fall back to reflecting over operand types for op_Addition method
}
```

… with this static method in the IntOps class:

```
object Add(int x, object y) {
    if (y is int) return x + (int)y; // modulo overflow handling
    if (y is double) return (double)x + (double)y;
    // ... lots more special cases ...
}
```

This was unfortunate for two reasons. One problem was that it bloated the resulting IronPython assemblies with optimized fast-paths for thousands of cases (about 180kb in the DLL). The larger issue, however, was that this only optimized for classes were known when the fast-paths were fixed. Any classes with user-defined conversions end up falling back to the full search reflection case on each execution of "x + y".

By implementing an adaptive caching system that caches the implementations for the actual sets of types observed by a given dynamic call site, both the generally common cases and the cases specific to a given application can be fast.

### 4.1.2 Language Interoperability

Assuming you were willing to do the optimizations discussed above, it would be possible to create a high-performing .NET dynamic language today without the DLR, as the DLR is simply an API that sits on top of the CLR and does not require any new built-in CLR features. This is, in fact, what the first versions of IronPython did, directly compiling Python code to IL that called run-time helper methods defined by IronPython. As the IronPython compiler knew about the semantics of standard .NET types, it could also provide a great experience targeting the BCL and existing C#/VB assemblies as well.

However, this story breaks down when you want an IronPython object to call into, for example, an IronRuby library and dynamically invoke functions on the objects it gets back. With no standard mechanism to understand how to perform dynamic operations on an external dynamic object, IronPython could only treat the object as a .NET object, accessing its statically-known API. Most likely any visible methods will be an implementation detail of IronPython rather than a representation of the user's intent.

The DLR provides for this common currency when calling into objects defined in other dynamic languages by establishing a protocol for those implementing dynamic objects. As a language creator or library author who wants other DLR-aware languages to consume your objects dynamically, you must have the objects implement IDynamicMetaObjectProvider, offering up appropriate implementations of DynamicMetaObject when IDynamicMetaObjectProvide's GetDynamicMetaObject() is called.

### 4.1.3 Creating Dynamic Call Sites

At a high level there are two ways to create dynamic call sites. A call site encapsulates a language's binder and the dynamic expression or operation to perform. A regular .NET language compiler (for example, what C# does for its 'dynamic' feature) emits a call site where dynamic expressions occur. The language can create the ahead of time, stash them in a functions constants pool, put them in static types, or whatever. The compiler then emits a call to the site's Target delegate to invoke the dynamic operation.

Dynamic languages built completely on the DLR, such as IronPython, use DynamicExpression nodes from Expression Trees v2 (which the DLR ships in .NET 4.0). The Expression.Compile method builds the call site objects and emits the right calls on their Target delegate when compiling.

## 4.2 Rules

Each dynamic call site keeps track how to perform the operation it represents. It learns different ways to perform the operation depending on the data (operands) that flows into the site. The binder provides these implementations of the operation along with restrictions as to when the site can correctly use the implementations. We refer to these restrictions and implementations together as rules. The call site compiles the rules it sees into a target delegate that embodies the site' cache.

To explore rules, let's consider the case of adding two dynamic operands in a DLR-aware language:

```
d1 + d2
```

Let's assume that the first time we hit this expression, both operands have a runtime type of int. The operations' dynamic call site has not yet bound any implementations for this addition site, and so it will immediately ask the runtime binder what to do. In this case, the runtime binder examines the types of the operands and decides to ensure that each operand is an int and then performs addition. It therefore returns an expression tree representing this:

```
(int)d1 + (int)d2
```

This expression tree serves as the **implementation**, which is the specific meaning the runtime binder has given to this dynamic expression in this exact situation. This expression tree alone would be sufficient to finish evaluating this expression, as it could be wrapped in a lambda expression, compiled into a delegate, and executed on the given operands. However, to implement a cache we also need to understand the extent of the situations in which we can reuse this implementation in the future. Is this implementation applicable whenever both operands are typed as int, or must the operands have not just the type int but perhaps an exact values encountered by this expression? The conditions under which an implementation applies in the future are known as the **test**.

For example, in the case above, the test returned by the binder might say that this implementation applies whenever the types of both arguments are int. This can be thought of as an if-condition that wraps the implementation:

```
if (d1 is int && d2 is int) {
    return (int)d1 + (int)d2;
}
```

By providing this implementation and this test, the binder is saying that in any case where d1 and d2 both have the runtime type int, the correct implementation will always be to cast d1 and d2 to int and add the results. This combination of a test, plus an implementation that is applicable whenever that test is met is what forms a **rule**. These rules are represented in the DLR as compound expressions formed from the implementation expression and the test.

A rule may also be generated for a failed binding to optimize future calls where it's guaranteed to fail again. For example, if the left operand was of type MyClass and this class does not have a user-defined + operator, the following rule may be returned:

```
if (d1 != null && d1.GetType() == typeof(MyClass)) {
      throw new InvalidOperationException("Runtime binding failed");
}
```

This rule chooses to test type identity so that it's only applicable when the type is *exactly* MyClass, as an instance of a derived class may derive its own + operator. If the language was such that it could be known that no derived class is allowed to define a + operator, or perhaps that this type is sealed, the binder could return a more efficient is-test. This is the type of decision the language binder makes as it is the arbiter of its language's semantics.

Languages with highly dynamic objects may define more advanced tests as well. For example, if an object allows methods to be added or removed at runtime, it's not correct to simply cache what to do when that method is invoked for some operand types, as the method may be different or missing on the next invocation. In this case, the object may choose to keep track of a version number, incrementing the version each time a method is added or removed from the object. The test would then check both that the parameter types match, and that the version number is still the same. In the case that the version number has changed, the test would fail, and this rule would no longer apply. The site would then call the binder to bind again.

## 4.3 Binders and CallSiteBinder

The component of a dynamic site that actually accepts an operation's arguments at runtime and produces rules is known as a **binder**. A binder encapsulates the runtime semantics of the language that emitted the call site as well as any information the language needs at run time to bind the operation. Each dynamic call site the compiler emits has an instance of a binder. There are several kinds of binders for different kinds of operations, each deriving from **CallSiteBinder**. At runtime, this class will perform the binding the compiler is skipping during compile-time.

An instance of a binder should encode all statically available information that distinguishes this operation from a similar operation. For our addition example, the language may define a general OperationBinder class, and then specify when constructing an instance of OperationBinder that the site's specific operation is addition.

It is up to the language to decide the specific static information it will need at runtime in addition to the list of arguments to bind this operation accurately. For a method invocation, say `d.Foo(bar, 123)`, the binder instance encodes the method name Foo, but may also choose to encode facts like the following if they are material to binding this invocation at runtime:

- the second parameter was passed a literal value
- named argument-passing was not used
- the entire expression was inside a checked-arithmetic block

It is entirely up to each language what compile-time information it chooses to encode in its binder for a given operation.

See the sites-binders-dynobj-interop.doc document for details on binders and API reference. See the section below on L2 Cache for high-level discussion of placing unique binder instances on multiple call sites when the binding metadata in the binders is equivalent.

## 4.4 CallSite<T> and Caching

When a compiler emits a dynamic call site, it must first generate a **CallSite<T>** object by calling the static method CallSite<T>.Create. The compiler passes in the language-specific binder it wants this call site to use to bind operations at runtime. The T in the CallSite<T> is the delegate type that provides the signature for the site's target delegate that holds the compiled rule cache. T's delegate type often just takes Object as the type of each of its arguments and its return value, as a call site may encounter or return various types. However, further optimization is possible in more restricted situations by using more specific types. For example, in the expression `a > 3`, a compiler can encode in the delegate signature that the right argument is fixed as int, and the return type is fixed as bool, for any possible value of a.

To allow for more advanced caching behavior at a given dynamic call site, as well as caching of rules across similar dynamic call sites, there are three distinct **caching levels** used, known as the L0, L1, and L2 caches. The code emitted at a dynamic site searches the L0 cache by invoking the site's Target delegate, which will fall back to the L1 and L2 caches upon a cache miss. Only if all three caches miss will the site call the runtime binder to bind the operation.

### 4.4.1 L0 Cache: CallSite's Target Delegate

The core of the caching system is the **Target** delegate on each dynamic site, also called the site's **L0 cache**, which points to a compiled dynamic method. This method implements the dynamic site's current caching strategy by baking some subset of the rules the dynamic site has seen into a single pre-compiled method. Each dynamic site also contains an **Update** delegate which is responsible for handling L0 cache misses by continuing on to search the L1 and L2 caches, eventually falling back to the runtime binder. The current method referenced by the Target delegate always contains a call to Update in case the rule does not match.

For example, an L0 Target method for a call site that has most recently adding ints might look like this:

```
if (d1 is int && d2 is int) {
    return (int)d1 + (int)d2;
}
return site.Update(site, d1, d2);
```

Note that the CLR's JIT will compile this down to an extremely tight method (for example, the `is` tests should be just a few machine instructions, and the casts will disappear). This means that while it's somewhat expensive to update the L0 cache for a new set of encountered types (as this requires compiling a new dynamic method), for L0 cache hits execution will come very close to the performance of static code.

### 4.4.2 L1 Cache: CallSite's Rule Set

As it takes time to generate the dynamic methods needed for the L0 cache, we don't want to throw them away when a new set of types is encountered. For this reason, a call site will keep track of the 10 most recently seen dynamic methods that it has generated in its **L1 cache**. When there is an L0 cache miss, and the Update method is called, each dynamic method in the L1 cache is called in turn until a hit is found. When a hit is found, the L0 cache is updated to point to this method once again.

#### 4.4.2.1 L2 Cache: Combined Rule Sets of All Equivalent CallSites

Each individual dynamic call site has a delegate type and a specific binder instance. The **L2 cache** is maintained across dynamic call sites with the same binder instance. The L2 cache allows the sitesto share rules via the binder instance.. The L2 cache helps eliminate startup costs for call sites that perform operations similar to those already performed elsewhere. Since the L2 cache currently holds 100 delegates, it also helps improve the performance of highly polymorphic sites that encounter many various types. Producers of binders must put canonical binder instances for a set of given metadata so that semantically equivalent call sties have the same binder instance on them.

Upon an L1 cache miss the binder would otherwise be required to generate a new dynamic method, either because this particular call site has not yet bound that operation, or because it has fallen off the end of this site's L1 cache. The shared L2 cache solves both of these problems by providing a larger cache that retains the rules seen by all equivalent call sites. An L2 hit means that the rule can be added back to the site's L1 cache and L0 target delegate without regenerating the dynamic method.

### 4.4.3 Other Optimizations

The general architecture of the DLR's call sites and rules is open to several optimization techniques. For example, the rule expression generated for a given execution of a call site may follow a similar template as the last rule generated, differing only in some ConstantExpression nodes. The DLR can recognize rules that are similar in this way and combine them into one compiled rule that replaces each differing ConstantExpression with a ParameterExpression that expects the value as a parameter. This saves the cost of rebuilding and re-JIT'ing the Target delegate as the constant changes. This is one example of various techniques call sites allow us to employ in the DLR.

## 4.5 DynamicObject and ExpandoObject

We make life much simpler for library authors who want to create objects in static languages so that the object can behave dynamically and participate in the interoperability and performance of dynamic call sites. Library authors can avoid the full power of DynamicMetaObjects, but they can also employ DynamicMetaObjects if they wish. The DLR provides two higher level abstractions over DynamicMetaObject: DynamicObject and ExpandoObject. For most APIs, these objects provide more than enough performance and flexibility to expose your functionality to dynamic hosts.

### 4.5.1 DynamicObject

The simplest way to give your own class custom dynamic dispatch semantics is to derive from the **DynamicObject** base class. DynamicObject lets your objects fully participate in the dynamic object interoperability protocol, supporting the full set of operations available to objects that provide their own custom DynamicMetaObjects. DynamicObject lets you choose which operations to implement, and allows you to implement them much more easily than a language implementer who uses DynamicMetaObject directly.

DynamicObject provides a set of 12 virtual methods, each corresponding to a Bind... method defined on DynamicMetaObject. These methods represent the dynamic operations others may perform on your objects:

```
public abstract class DynamicObject : IDynamicMetaObjectProvider
{
    public virtual bool TryGetMember(GetMemberBinder binder,
                                     out object result)
    public virtual bool TrySetMember(SetMemberBinder binder,
                                     object value)
    public virtual bool TryDeleteMember(DeleteMemberBinder
binder)

    public virtual bool TryConvert(ConvertBinder binder,
                                   out object result)
    public virtual bool TryUnaryOperation
        (UnaryOperationBinder binder, out object result)
    public virtual bool TryBinaryOperation
        (BinaryOperationBinder binder, object arg,
         out object result)

    public virtual bool TryInvoke
        (InvokeBinder binder, object[] args, out object result)
    public virtual bool TryInvokeMember
        (InvokeMemberBinder binder, object[] args,
         out object result)
    public virtual bool TryCreateInstance
        (CreateInstanceBinder binder, object[] args,
         out object result)
    public virtual bool TryGetIndex
        (GetIndexBinder binder, object[] args, out object result)
    public virtual bool TrySetIndex
        (SetIndexBinder binder, object[] indexes, object value)
    public virtual bool TryDeleteIndex
        (DeleteIndexBinder binder, object[] indexes)
```

You could have your own TryGetMember implementation look up “Foo” in a dictionary, crawl through a dynamic model like XML, make a web request for a value, or some other custom operation. To do so, you would override the TryGetMember method and just implement whatever custom action you want to expose through member evaluation syntax. You return true from the method to indicate that your implementation has handled this situation, and supply the value you want returned as the out parameter, result.

By default, the methods that you don’t override on DynamicObject fall back to the language binder to do binding, offering no special behavior themselves. For example, let’s say you have a

class, MyClass, derived from DynamicObject that does *not* override TryGetMember. You also have an instance of MyClass in a variable, myObject, of type C# 'dynamic'. If you evaluate `myObject.Foo`, the evaluation falls back to C#'s runtime binder, which will simply look for a field or property named Foo (using .NET reflection) defined as a member of MyClass. If there is none, the C# binder will store a binding in the cache that will throw a runtime binder exception in this situation. A more real example is that you do override TryGetMember to look in your dictionary of dynamic members, and you return false if you have no such members. The DynamicMetaObject for MyClass produces a rule that first looks for static members on MyClass, then calls TryGetMember which may return false, and finally throws a language-specific exception when it finds no members.

In the full glory of the interoperability protocol, a dynamic object implements IDynamicMetaObjectProvider and returns a DynamicMetaObject to represent the dynamic view of the object at hand. The DynamicMetaObject looks a lot like DynamicObject, but its methods have to return Expression Trees that plug directly into the DLR's dynamic caching mechanisms. This gives you a great deal of power, and the ability to squeeze out some extra efficiency, while DynamicObject gives you nearly the same power in a form much simpler to consume. With DynamicObject, you simply override methods for the dynamic operations in which your dynamic object should participate. The DLR automatically creates a DynamicMetaObject for your DynamicObject. This DynamicMetaObject creates Expression Trees (for the DLR's caching system) that simply call your overridden DynamicObject methods.

### 4.5.2 ExpandoObject

The **ExpandoObject** class is an efficient implementation of a dynamic property bag provided for you by the DLR. It allows you to dynamically retrieve and set its member values, adding new members per instance as needed at runtime. Because ExpandoObject implements the standard DLR interface IDynamicMetaObjectProvider, it is portable between DLR-aware languages. You can create an instance of an ExpandoObject in C#, give its members specific values and functions, and pass it on to an IronPython function, which can then evaluate and invoke its members as if it was a standard Python object. ExpandoObject is a useful library class when you need a reliable, plain-vanilla dynamic object.

To allow easy enumeration of its values, ExpandoObject implements IDictionary<String, Object>. Casting an ExpandoObject to this interface will allow you to enumerate its Keys and Values collections as you could with a standard Dictionary<String, Object> object. This can be useful when your key value is specified in a string variable and thus you cannot specify the member name at compile-time.

ExpandoObject also implements INotifyPropertyChanging, raising a PropertyChanging event whenever a member is modified. This allows ExpandoObject to work well with WPF data-binding and other environments that need to know when the contents of the ExpandoObject change.

## 4.6 Default Binder -- Runtime Utility for Language Implementers

The DefaultBinder provides a simple way for languages that do not have deep specific .NET binding semantics to get started. It provides a number of hooks that allow the language to

partially customize the Defaultbinder to their specific behavior. The default binder supports the following on static .NET objects:

- Conversions
- creating new instances
- getting/setting members (fields, properties, methods)
- invoking objects (delegates or other types that define a specially named call method)
- performing operations (addition, subtraction, etc.)
- binding to a selection of overloaded methods

Languages use the DefaultBinder by deriving from it. From there they can start overriding virtual methods to customize the behavior. The first function typically to override is GetMember. This method provides the DLR GetMemberBinder (code is slightly out of date here), which has the information the language needs such as the .NET type and member name. The language can use this information to return the member's value. The default binder uses the GetMember method for all of its member resolution. Thus, languages can

- manufacture members/methods that don't really exist on static .NET types (such as meta-programming hooks)
- implement extension methods
- filter out members from .NET types (for example, Python could hide String.Trim this way and manufacture the Python strip method)

The next functionality languages usually override is overload resolution. The default binder has two different mechanisms for overload resolution which we'll merg. Languages get overload resolution that works similar to C#'s but that's also customized for their rules. For example, a language might want to allow passing bools where functions take integers. The language could indicate that bools are convertible to ints and provide conversion logic when the call happens. Much more is possible of course.

The overload resolution uses DynamicMetaObjects. The methods on DefaultBinder take DynamicMetaObjects and return one with the appropriate restrictions for invoking the chosen overload. For example, if there two overloads with two positional arguments, both typed to object, there is no need to do a type check on the argument at that position.

Finally the language implementer would probably customize the results of failures. Errors should be raised as they are expected in that language. The default binder provides a number of hooks for the language to produce the Expression Tree that embodies the semantics of what the language would do in a particular situation. For example, if a member doesn't exist, Python wants to throw an exception, but JS wants to return it's sentinel $Undefined value. Another example is when an error message is actually reported. Python would like to report the error in terms of Python type names (str, int, long, float, etc.), but other languages would want their own type names (or maybe the standard .NET names).

# 5 Language Implementation

It is hard to separate language implementation concepts from the runtime concepts with dynamic languages. However, we try to do so by defining the language implementation aspects of the DLR as are shared ASTs (Expression Trees), LanguageContext, language interop with IDynamicMetaObjectProvider, compilation, and utilities.

## 5.1 Expression Trees

In the .NET Framework 3.5 we created Expression Trees (ETs) to model code for LINQ expressions in C# and VB. They were limited in .NET 3.5 to focus on LINQ requirements; for example, a LambdaExpression could not contain control flow, only a simple expression as its body. Looking forward, there are several reasons we'd like to extend ETs:

- We'd like to further develop a single semantic code model as a common currency for compilation tools.
- We're adding the Dynamic Language Runtime (DLR) into the .NET Framework, and it uses semantic trees to represent code as a means for making it easier to port new dynamic languages to .NET.
- The DLR also uses semantic trees to represent how to perform abstract operations in its dynamic call site caching mechanism.
- We want to support meta-programming going forward where customers can more readily get programs as data, manipulate the data, and emit new code as data based on the input code.

Obviously, to support the above goals, we need more in the ET model than v1 provided. We need to model control flow, assignment, recursion, etc., in addition to simple expressions. There is no plan in .NET 4.0 to add modeling for types/declarations, but we'll consider these for V-next+1 (that is, the next major release after .NET 4.0).

Some quick terminology:

- The term "ET" indicates a tree structure of instances of Expression (direct or indirect).
- The term "ET node" indicates a single instance of Expression (direct or indirect). The ET node could be the root of a tree.
- The term "ET node type" means the specific class of which the ET node is an instance.
- The term "ET kind" refers to the Expression.NodeType property or indicates the value of the ExpressionType enum. This is a legacy naming issue.
- The term "sub ET" indicates the root of a tree where the root is an interior or leaf node of some other ET.

The following sub sections are some highlighted concepts for Expression Trees v2.

### 5.1.1 Expression-based Model

One high-order bit to language design is whether to be expression-based. Do you have distinct notions of statements or control flow, or do you have a common concept of evaluating expressions where everything has a value. We decided to stay expression-based, so statements are modeled as having a result value and type.

There are several reasons for this design:

- Expression remains the base type for all ET nodes, and we avoid dual type hierarchies.
- Void is already allowed as a type, indicating there is no return value for an expression.
- Lambdas don’t change at all from v1 to v2.

- Being expression-based matches many languages (Lisp, Scheme, Ruby, F#), and it does no harm when modeling other languages. They can easily make expressions be void returning.

Let's look at a couple of examples:

- BlockExpression has a value. By default its value is the last expression in the sequence, and its type is the same type as the last expression. This design also allows us to avoid another CommaExpression since BlockExpression now models this semantics as well.
- We model 'if' with ConditionalExpression, which returns the value of its consequent or alternative expression, whichever executes. The types of the branches must match the type of the ConditionalExpression. If there's no alternative expression, then we use a DefaultExpression with the matching type. Languages with distinct notions for statements often have a 'e1 ? e2 : e3' expression since their 'if' cannot return values, but they can model this with ConditionalExpression.

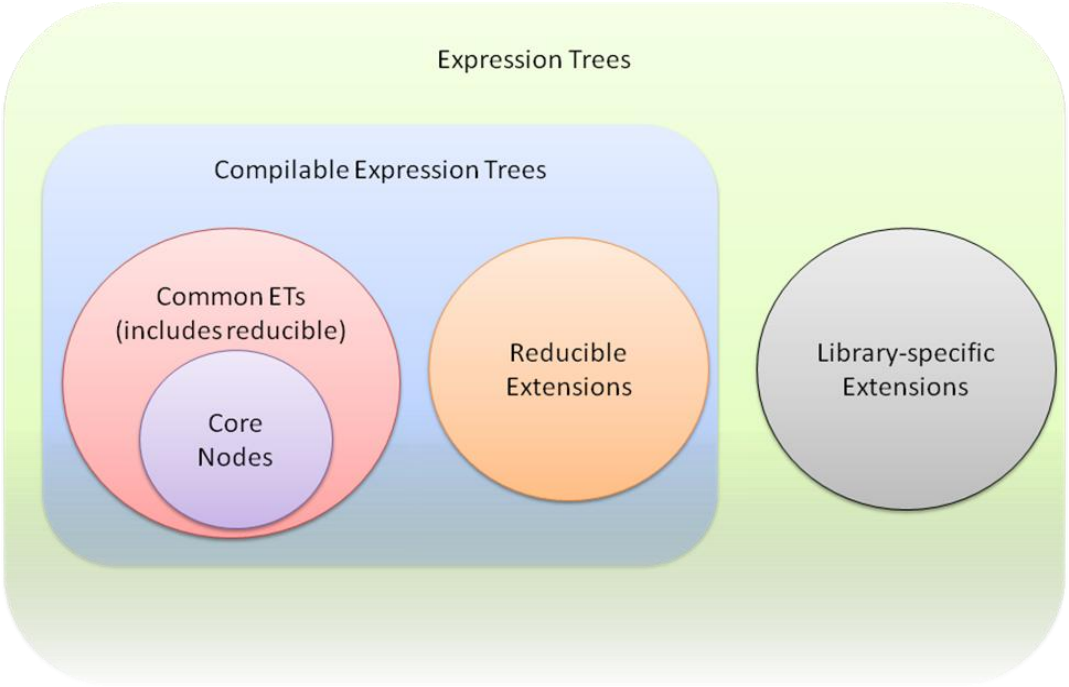
DefaultExpression serves two useful purposes in our expression-based model. First the Expression.Empty factory returns a DefaultExpression with Type void. This can be useful if you need an expression in a value-resulting position that matches the containing expressions result type. The second use of DefaultExpression is when you do have a non-void Expression in which you do need to sometimes return the "`default(T)`" value. Without this expression, you would have to generate a lot more ET to express "`default(T)`".

Often you do not need to use Expression.Empty to match a containing node's result type. There are expressions used in common patterns, typically control flow expression, where the result value is not used. For these common patterns, some nodes implicitly convert to void or squelch a result value. SwitchExpression, ConditionalExpression, TryExpression, BlockExpression, LambdaExpression, GotoExpression, and LabelExpression all automatically convert their result expression to void if they themselves have a void Type property (or delegate with result type in the case of lambdas).

### 5.1.2 Reducible Nodes

We have a tension for what level of abstraction to provide in our model (see expr-tree-spec.doc on codeplex.com/dlr). While we think Design Time Language Models have a distinct mission from Expression Trees v2, we would like to allow for smooth interoperability between them. We enable higher-level models (even language-specific models) that can reduce to a common set of ET v2 node types that all consumers can process. Programs can query an ET node as to whether it reduces, and if so, a program can call on the node to reduce itself. When a node reduces, it returns a semantically equivalent ET with a root node than can replace the original ET node.

Reductions are allowed to be partial. The resulting ET may include nodes that need to be further reduced. Typically the immediate result of reducing one node comprises only children that are common ET v2 node types. Expression.Compile only compiles common node types and those that reduce to common nodes. If a node type does not reduce or does not reduce to only common nodes, then it may still be useful as part of a library-specific set of extensions that model code. An example might be a Design Time Language Model for code that either has too many errors to reduce or that is part of a model too specific to tooling needs to bother reducing it to common nodes.




The ET v2 common set of nodes will include some reducible nodes. For example, for meta-programming goals, there will be higher-level iteration models. We'll include ForExpression for counter termination, ForEachExpression for member iterations, WhileExpression for test first iterations, and RepeatUntilExpression for test last iterations. We'll also include a fundamental LoopExpression to which the other iteration models reduce. Other examples include BinaryExpression with node kind AddAssign (compound assignment) or UnaryExpression with node kinds PreIncrementAssign and PostIncrementAssign.

*Note, due to time constraints we cut the higher-level iteration node types for .NET 4.0, but they will likely show up soon in the DLR's codeplex open source project.*

Common ET nodes with a given node kind are either reducible always, or never. That is, a node is not conditionally reducible based on other properties it has that may be different for different instantiations. For example, the GeneratorExpression nodes are always reducible. Regardless of the reducibility, the compiler may have direct support for the node kind, or the compiler may reduce the nodes. For example, when we add ForEachExpression, the compiler will likely directly compile it without reducing it.

### 5.1.3 Bound, Unbound, and Dynamic Nodes

There are three categories or states of being bound for modeling expressions. More commonly mathematicians or computer scientists think of only two, bound and unbound. For example, in the expression "for all x such that 0 < x + y < 10", 'x' is a bound variable while 'y' is a free reference or unbound variable. If 'y' were not present in the expression, the expression would be fully statically bound such that we could evaluate it. However, to evaluate the expression, we need to first bind 'y' to some value.

**An unbound ET node:**

- would need to be bound before executing it

- would represent syntax more than semantics
- would have a Type property that is null (see .NET 4.0 vs. V-next+1 note below)

Consider a language that supported LINQ-like expression and that also had late-bound member access (for example, if VB added late-bound LINQ). You would then need to model unbound trees for the lambda expression in the following pseudo-code:

```
o.Where( lambda (x) => x > 0 )   #o had late bound semantics
```

To be able to execute an ET modeling this code, you would need to inspect the runtime type of 'o', search its 'Where' overloads, and pattern match for one that can take a delegate. Furthermore, you would need to match lambda expression to the delegate. The delegate needs take an argument and returns a value with some type. The delegate's type for 'x' needs to make sense to bind '>' to an implementation taking the type of 'x', an operand assignable from integer, and returning the type of the delegate.

A key observation in this situation is that the late-bound node representing the call to 'Where' necessarily has language-specific binding information representing the lambda. The representation cannot be language-neutral semantically. It also can't even be just syntax in any common representation because you need the language that produced the ET to process the lambda representation in the presence of runtime type information while binding. Support for unbound ETs may not be a good solution or one worth trying to share across languages.

**NOTE:** Since no languages currently support late bound LINQ expressions, we won't actually allow Expression.Type to be null in .NET 4.0. We'll reconsider this in V-next+1 if we think ETs are useful to languages that need to represent unbound trees like the lambda expression in the example above.

**In the ET v2 model, a bound ET node:**

- has non-null Type property (that is, we statically know its type)
- could be dynamic expression

A dynamic expression often has a Type property that is Object, but its Type that is not null. It might not be Object as well. For example, in "`if x > y`" the ET node for '>' could be typed Boolean even if it is a dynamic node.

**The ET v2 model includes dynamically bound nodes that:**

- must be resolved at run time to determine how to perform the operation they model
- represented by DynamicExpression nodes

#### 5.1.3.1 DynamicExpression Node

The DynamicExpression has binding information representing the metadata that further describes the expression beyond the ET node type and any children it has. For example, an ET representing the dynamic expression for comparing a variable to a string literal might have a flag in its binding information indicating whether the compilation unit had an option set to compare strings case-sensitively (which VB has). The binding information also encapsulates the language that created the ET node. The language determines the semantics of the dynamic expression at runtime when its binder searches for an implementation of the operation represented by the node.

#### 5.1.3.2 Binding Information in Expression Nodes

Expression nodes can have semantic hints or specifications that are more detailed than, for example, BinaryExpression Add on Int32s or MethodCallExpression with instance and name.

These nodes can have MethodInfos attached to them to indicate the exact implementation of Add or resolution of method name to invoke. The MethodInfos serve two main purposes. The first is to be very exact when creating a node what the implementation is for the node's semantics. Language implementations should supply MethodInfos rather than leaving method resolution to the Expression factories because those factories may not resolve overloads in the same way the language would. The second job of the MethodInfos is to provide hints to LINQ providers that might interpret ETs. Those providers can have tables mapping to implementations of the node's semantics when they aren't actually compiling the ET and executing in a .NET run time.

This extra information is required with DynamicExpression nodes. They must have binding information that can be emitted when compiling. The binding information informs the run-time binder how to search for a correct implementation of the node's semantics, given the run-time operands that flow into the operation represented by the ET node. ETs use a CallSiteBinder as the binding information representation, not MethodInfo. In fact, the CallSiteBinder is also the run-time object used in the DLR's CallSites that manage the fast dynamic dispatch for dynamic operations. CallSiteBinder encapsulates both the binding information for the exact semantics of the node and the language that created the node, which governs the binding of the operation at run time.

Two design issues arise immediately from the choice to use CallSiteBinders vs. MethodInfos. The first is serializability. The ET design supports fully serializable ETs but doesn't enforce that they are always serializable. One reason we use the CallSiteBinders in the DynamicExpression is that they naturally fit exactly what the binding information is that the ET needs and help in hosted execution scenarios. If a language produces an ET as part of a hosting scenario to immediately execute the ET, then the binder can tuck away pointers to live data structures used by the language's engine. Languages can still produce ETs with DynamicExpressions that are serializable if they need to do so.

The second design issue is re-using MethodInfos by deriving custom implementations for use with DynamicExpression nodes. There's a nice consistency in representing the binding information as a MethodInfo, looking on at ETs only. However, the roles played by the MethodInfo are different than the binding information on DynamicExpression. It is more important to have the dynamic binding information be consistent across ETs, the DLR fast dynamic CallSites, and the DLR interoperability MetaObject protocol. Not only does MethodInfo have many members that would throw exceptions if used for dynamic binding information, it would be awkward to require creating an LCG method so that the MethodInfo was invocable. CallSiteBinders are best for detailed semantics capture in DynamicExpression nodes, not MethodInfos.

### 5.1.4 Iteration, Goto's, and Exits

Representing iteration has a couple of interesting concepts and design points. How to represent iteration goes to the heart of what abstraction level to model for expressions (see expr-tree-spec.doc on codeplex.com/dlr). Whether to include Goto goes back ages in language design. ETs v2 provides a nice combination of high-level modeling and lower-level support if needed. There are some higher level modeling nodes so that you almost never need Goto, but we provide Goto for reducing some node types to more primitive control flow.

We provide GotoExpression because it is needed for C#, VB, and other languages. The ET v2 GotoExpression started with simple, clean semantics designed into C#. Basically this meant that the target label of the Goto must be lexically within the same function body, and you could only exit inner basic blocks to outer basic blocks. However, we need to accommodate more of VB's older Goto semantics, and we need a richer Goto for some of the ET transformations we do internally (see section 5.1.4.3).

GotoExpression has an optional Value expression. This allows GotoExpression to enable other node types to truly be expressions. The value expression's type must be ref assignable with the type of the GotoExpression's label target. See the following sub sections for more details.

GotoExpression has a Kind property with a value from GotoExpressionKind (Goto, Break, Continue, Return). This is explained further in the following sub sections. This is convenience for meta-programming purposes.

#### 5.1.4.1 LoopExpression and Higher-level Iteration Node Types

Given Goto, we provide a basic LoopExpression with explicit label targets for where to break to and where to continue to. Explict labels in LoopExpressions have multiple benefits. You can use GotoExpression inside the LoopExpression's Body expression and verify the jumps are to the right locations. Explicit labels support languages that can return from or break out of outer loops or scopes, such as JScript or Lisp. Explicit label targets make transformations easier to get right and allow for better error reporting when transformations are not right.

For meta-programming goals, there will be higher-level iteration models. We'll include ForExpression for counter termination, ForExachExpression for member iterations, WhileExpression for test first iterations, and RepeatUntilExpression for test last iterations. These will be common nodes that reduce to combinations of LoopExpression, BlockExpression, GotoExpression, etc.

*Due to time constraints we cut the higher-level iteration node types for .NET 4.0, but they will likely show up soon in the DLR's codeplex open source project.*

#### 5.1.4.2 Modeling Exits, LabelExpression, and GotoExpression with a Value

As many have observed, lexical exits are ultimately just a Goto, proceeding to the end of the function and then returning. Sometimes when you leave a function, you leave one or more values on the stack. The ET model only supports a single return value inherently. Once we had to have Goto, there is an economy of design obtained by merging the models of explicit function return and Goto.

There are a couple of very nice benefits from merging lexical exits and Goto. The biggest is that by having Goto optionally carry a value to its target, more of our nodes became truly expression-based. This makes the overall model more consistent by more fully embracing the benefits of being expression-based. For example, LoopExpression is truly an expression that can be used anywhere because you can exit the loop with a value at the break label target.

The second benefit of merging Goto with lexical exits is that we've added explicit label targets to some nodes. This enables more Goto checking in factories and better error messages when compiling. For example, when you have a Goto expression inside a LoopExpression with kind Break, the expression compiler can ensure the Goto's target is the containing Loop's break label target. Also, when you form a GotoExpression, you have to supply the label target which has a

Type property. The factory can ensure the Goto and the label target match by type, and the compiler can check the types match the type of the containing expression. Having label targets explicit in the tree also enables tree transformations to be safer and more reliable.

One quirk in the design is how to handle LabelExpression which has a LabelTarget that marks a destination in code for Goto. If a LabelExpression has non-void type, and execution gets to the target via Goto with a value of that type, we're consistent. What happens if I encounter the target location by straight line sequence, and how do we keep the IL value stack consistent? We solve this by adding an optional default value expression to a LabelExpression and by specifying the semantics of LabelExpression to place its target AFTER the default expression. In practice this works very naturally. For example, a LambdaExpression's Body can be a LabelExpression whose default value expression is the actual body of the function. Then the actual body of the function, an expression, is the value left on the stack if you naturally flow through the expressions in the body. If you exit via a GotoExpression (with node kind Return), you have a verifiable target at the end of the function and a verifiable value type that matches the return type from the Type property of the LambdaExpression's.

#### 5.1.4.3 GotoExpression Capabilities

As stated, we expanded Goto capabilities beyond C#'s. VB is not fully using the DLR yet, but when it does, we will need a more flexible Goto. If we do not allow more cases for GotoExpression, VB would need to produce a VBBlockExpression and their own VBGotoExpression that reduced to a complicated rewriting of the ET. It seems useful to provide the more general GotoExpression. However, VB would still need a special VBBlock to model their on_error_goto semantics, which seems too specific to a single language to generally model in common ET nodes.

ETs v2 limit Goto lexically within a function. ETs allow jumping into and out of the following:

- BlockExpressions
- ConditionalExpressions
- LoopExpressions
- SwitchExpressions
- TryExpressions (under certain situations)
- LabelExpressions

ETs allow some jumps relative to TryExpressions:

- jumping out of TryExpression’s body
- jumping out of a CatchBlock’s body
- jumping into TryExpression’s body from one of its own CatchBlocks

The above constitutes what ETs v2 allow. Just by way of examples, we do not allow jumping into the middle of argument expressions, such as those in binary operands, method calls, invocations, indexing, instance creation, etc. We do not allow jumping into or out of GeneratorExpressions. You could however jump within a BlockExpression used as an argument expression.

### 5.1.5 Assignments and L-values

We model assignments with BinaryExpression nodes that have an Assign node kind. The factory methods restrict the Left expression of the BinaryExpression to a fixed set of common ET node

types that the compiler recognizes. This permitted ET node types are ParameterExpressions, MemberExpressions, and IndexExpressions.

For ref and out parameters, we also restrict ET node types, but we do allow a few more node types. We additionally allow BinaryExpressions with node kind ArrayIndex, MethodCallExpressions with MethodInfo that is Array.Get, and the new UnboxExpression. The first two are legacy from LINQ ETs v1, and we will obsolete them eventually (see section 5.1.5.1). Expression.Compile handles write backs for these expressions passed to ref and out parameters. We explicitly do not support property accesses wrapped in conversion expressions due to ambiguities with how to convert back when writing back the out or ref value.

*We believe in V-next+1 we can introduce a generalized l-value model. We would add pre and post expressions for temps set up and for write backs. We don't think this model would be overly complex. If added, then l-value positions could be any node type that was writable and had pre and post ETs for setting up temps and writing back values. We would still NOT support l-value positions with property accesses wrapped in conversion expressions.*

#### 5.1.5.1 Indexed Locations

In ETs v2 we have IndexExpression that handles array access, indexers, and indexed properties. You can use these as l-values for assignments and ref/out parameters.

We will obsolete the LINQ v1 support for BinaryExpressions with node kind ArrayIndex and MethodCallExpressions with methodinfo Array.Get. We only support those for ref/out parameters for LINQ v1 compatibility, and we do not support them for the new BinaryExpression with node kind Assign. The plan, which could change, is as follows:

- In .NET 4.0, add bold red text to MSDN docs around ArrayIndex factories. Add bold red text to MethodCallExpression factories if methodinfo has ref/out parameters, and you're using one of the two nodes (BinaryExpressions with node kind ArrayIndex or MethodCallExpressions with methodinfo Array.Get).
- In v-next+1, mark the factory methods as obsolete.
- In v-next+1, issue warnings when compiling ref/out parameters whose expressions are BinaryExpressions with node kind ArrayIndex or MethodCallExpressions with methodinfo Array.Get.
- In v-next+2, remove ArrayIndex factories.
- In v-next+2, throw exceptions where we issues warnings in v-next+1.

We also disallow ref and out arguments for IndexExpression. Neither VB nor C# support these. They are not part of .NET CLS.

When creating IndexExpressions, you must supply PropertyInfo. Another clean up to the LINQ v1 trees is that you cannot use random get/set method pairs.

#### 5.1.5.2 Compound Assignment or In-place Binary and Unary Operations

We represent operations such as "+=" and "++" as BinaryExpression or UnaryExpression nodes with distinct node kinds (for example, AddAssign). These nodes are reducible common nodes. They reduce to the appropriate binary assignment expressions or blocks with temps to ensure sub-expressions are evaluated at most once. Regarding dynamic expressions at run time, some languages will need to inspect the l-value expression for .NET types to handle events, properties with delegates, etc., the right way when binding the expression.

#### 5.1.5.3 User-defined Setting Syntax

Languages that have user-defined setting forms for storable locations should provide language-specific reducible nodes. These nodes can reduce, for example, to a MethodCallExpression that takes the arguments defining the location and an argument for the value. We don't provide extensibility for this sort of language feature.

#### 5.1.5.4 Destructuring Assignment

Languages that support destructuring assignment should provide language-specific reducible nodes. Different languages handle destructuring bind in different ways (first-class carrier objects for the values, single vs. multiple kinds of carrier objects for values, runtime-internal carrier objects or calling conventions, etc.). A language-specific node will allow for better meta-programming consumers working with a given language. These nodes can reduce to an ET with exact semantics represented in common ET v2 nodes types.

### 5.1.6 Blocks, Scopes, Variables, ParameterExpression, and Explicit Lifting

ETs v2 model variable references with ParameterExpression nodes and a node kind value of Parameter. Initially for readability of code we chose to introduce a VariableExpression node type. It turns out in practice this meant much code that processed ETs had to be duplicated due to static typing in languages or had to use typeof() to dispatch to the right code if using Expression as a variable's type. Very little code actually needed to treat the declarations or references differently.

ETs v2 includes a BlockExpression node type that has an explicit list of variables. It creates a binding scope. Producers of ETs can use these to introduce new lexical variables, including temporaries. Blocks do not guarantee definite assignment. Languages need to do that when producing ETs. To initialize variables, BlockExpressions must explicitly include expressions in their body to set the variables. Definite assignment semantics is the concern of consumers of trees and compilers that enforce those semantics. Different languages have varying semantics here from definite assignment, to explicit unbound errors, to sentinel $Unassigned first class values. Lastly, recall ETs v2 is expression-based, so the value of a BlockExpression is the value of the last expression in its body.

Some languages have strong lexical semantics for unique binding of variables. For example, in Common Lisp or Scheme, each iteration around a loop or any basic block of code creates unique bindings for variables introduced in those basic blocks. Thus, returning a lambda would create unique closures environments for each iteration. Some languages, such as Python and F#, move all variables to the implicit scope of their containing function. ETs v2 supports both models, all depending on where you create the BlockExpression in the ET and list variables. For the stronger lexical model, for example with the loop, place the BlockExpression inside the loop body and list variables there, instead of putting the Block Expression outside the loop or at the start of the function.

ETs v2 also supports explicit lifting of variables to support languages that provide explicit meta-programming of local variables. For example, Python has a "locals" keyword that returns a dictionary of the lexical variables within a function so that you can manipulate them or 'eval' code against your function's environment. You can use the RuntimeVariablesExpression to list the ParameterExpressions that you need explicitly lifted.

### 5.1.7 Lambdas, Exits, and Result Types

Lambdas are modeled with LambdaExpression and Expression<T>. The latter derives from the former, and the T is a delegate type. LambdaExpression.Type holds the same T, and there is a ReturnType property that holds the type of value the T delegate would return. All lambdas created by the factory methods are actually Expression<T>. LambdaExpression provides the general base type for code that needs to process any lambda, or if you need to make a lambda with a computed delegate type at runtime. LambdaExpression supports two Compile methods that return a delegate of type LambdaExpression.Type, which can be invoked dynamically at run time.

Handling arbitrary returns or exits from lambdas has some interesting issues. Returns from a lambda can be arbitrarily deep in control constructs and blocks. Lexical exits represent a sort of non-local exit from nested control constructs. The expression that is the body of a lambda might have a particular type from the last sub expression it contains. Execution of the ET may never reach this last sub expression because of a Goto node. Even though we've added these sorts of control flow to ETs v2, they still keep the constraints ETs v1 had regarding LambdaExpression.Type and .Body.Type.

#### 5.1.7.1 Ensuring Returns are Lexical Exits with Matching Types

We model returns with GotoExpression and LabelExpression. Due to how ETs v2 use these, we have the same verifiable properties discussed in section 5.1.4.2. We can verify exits are lexical. We can verify that the function's return type, its Body's type, and any lexical exits with result values all match in type. At one point in the ETs v2 design, we were not able to ensure the types invariant, and while the v1 behavior was not guaranteed, we didn't want to break that. Note, there is one case from v1 that counters this invariant property of matching types. When the lambda's delegate type has a void return, the body's type does not have to match since any resulting value is implicitly converted to void or squelched.

There is a factory method to create a LambdaExpression that only takes an Expression for the lambda's body and a parameters array. This factory method infers the lambda's return type from the body expression. For many ET node types, we only ensure a parent's Type property matches the sub expressions' Types that by default produce the result of the parent node. For example, when constructing a BlockExpression, the factory only checks that the last sub expression has the same type as the BlockExpression. However, you can get proper checking on lexical exits from lambdas at creation time (without waiting to compile) using LabelExpression as the lambda's body.

The best pattern for creating lambdas with returns is to make the LambdaExpression's Body be a LabelExpression. The LabelExpression's default value expression is the actual body of the function. Then the actual body of the function, an expression, is the value left on the stack if you naturally flow through the expressions in the body. If you exit via a GotoExpression, you provide a value expression and a label target. This pattern enables ETs with a verifiable target at the end of the function and that any return value has the appropriate matching type.

You could use a BlockExpression as the LambdaExpression.Body. You would then have to put a LabelExpression as the last expression in the block. If the lambda returned non-void, you would need to make the BlockExpression.Type match the LambdaExpression.ReturnType. To do this, you would need to make the LabelExpression.Type match the block's type, and you would need to fill in the LabelExpression's default value expression with a DefaultExpression, supplying the

lambda's return type. This is the normal way to think about creating a target to use as the exit location for the function, but it is much more work than using a LabelExpression as the lambda's body itself.

When rewriting reducible nodes in a lambda, if you need to create a return from the lambda, you'll need to ensure the LambdaExpression's Body is a LabelExpression. If there isn't one already, your rewriting visitor will need to create the label target and then as you unwind in the visitor, replace the Body of the LambdaExpression with a LabelExpression using the same label target.

Lastly, what about languages that allow return targets with dynamic extent? Some languages allow closing over function return or block labels (that is, true non-local returns). If these languages do not require full continuations semantics, they could easily map these non-local return label closures to throws and catches that implement the semantics.

#### 5.1.7.2 Throw

We model Throw with a UnaryExpression. We had a ThrowExpression at one point. To be consistent with the "shape" aspect of re-using expression node types with the same kinds of children or properties, you can think of Throw as an operator with a single argument.

The Type property of the UnaryExpression with node kind Throw does not have to be void, which you may expect since the Throw never really returns. Using types other than void can be useful for ensuring a node has consistently typed children. For example, you might have a ConditionExpression with Type Foo, where the consequent returns a Foo, but the alternative is a Throw. Allowing the UnaryExpression with node kind Throw to have a non-Void Type, you do not have to artificially wrap your alternative in a BlockExpression and use a DefaultExpression in it.

CatchBlocks are helper objects that convey information to a TryExpression, like the MethodInfos or CallSiteBinder objects some Expression have. CatchBlock is not an expression as you might expect, primarily because they cannot appear anywhere any expression can appear. We could have thought of TryExpression as having a value from its Body, but sometimes its value comes from a CatchExpression. This would be analogous to ConditionalExpression. However, unlike a language like Lisp where Catch is a first-class expression, we felt the model we settled on is both expression-based and more amenable to .NET programmers.

#### 5.1.7.3 Recursion

ETs avoid needing a Y-Combinator expression by simply using assignment. An ET can have a BlockExpression with a variable assigned to a LambdaExpression. The LambdaExpression's body can have a MethodCallExpression that refers to a ParameterExpression that refers to the variable bound in the outer scope. The compile effectively closes over the variable to create the recursive function.

### 5.1.8 Generators (Codeplex only)

*Cut from .NET 4.0, available only on www.codeplex.com/dlr .*

Generators are first-class concepts in the ET v2 model. However, they are only available in the DLR's Codeplex project. The code can be re-used readily and ships in IronPython and IronRuby. The basic model is that you create the LambdaExpression with your enumerable return type and

then use a GeneratorExpression inside the lambda to get the yielding state machine. The outer lambda can do any argument validation needed, and the GeneratorExpression reduces to code that closes over the lambda's variables. The body of the GeneratorExpression can have YieldExpressions in it. The generator node reduces to an ET that open codes the state machine necessary for returning values and re-entering the state machine to re-establish any dynamic context (try-catch, etc.).

The main reason for not shipping generators in .NET 4.0 is they have a couple of features that are specific to Python, and we don't have time to abstract the design cleanly. Python allows yield from finally blocks, and while we could define what that means, we'd need offer some extensibility here; for example, what happens when the yield occurs via a call to Dispose. Python requires 'yield' to be able to pass a value from the iteration controller into the GeneratorExpression body, which the generator may use to change the iteration state.

### 5.1.9 Serializability

The ability to serialize an ET is important. You may want to send the ET as a language-neutral semantic representation of code to a remote execution environment or to another app domain. You might store pre-parsed code snippets as ETs as a representation higher level than MSIL and more readily executable than source. Nothing should prevent the ability to save an ET to disk and reconstitute it.

It will always be possible to create an ET that does not serialize. In fact, in the hosted language scenarios, most produced ETs may not serialize because they are created for immediate execution. In this case it is fine to directly point at data needed for execution that may not serialize. For example, DynamicExpression nodes can hold onto rich binding information for use at runtime. They may have references to the ScriptRuntime in which they execute or other execution context information.

If a language does need to refer to a ScriptRuntime or scope objects for free references to variables, then the language can still create serializable ETs. The entry point to executable code produced from an ET is always a lambda (even if it is an outer most lambda wrapping the top-level forms of a file of script code). The language can create the LambdaExpression with parameters for the ScriptRuntime and/or variables scope chain through ScriptRuntime.Globals. Since the language is in control when it invokes the lambda at a later time, or in another context, it can pass in the execution context the code needs. Finally, if the language uses DynamicExpression, it needs to ensure its CallSiteBinders are serializable.

### 5.1.10 Shared Visitor Support

ETs v2 provides a tree walker class you can derive from and customize. Customers often asked about tree walkers, and the DLR uses them a lot too. With the advent of Extension node kinds and reducibility, providing a walker model is even more important for saving work and having well-behaved extensions going forward. Without providing a walking mechanism out of the box, everyone would have to fully reduce extension nodes to walk them. Reducing is lossy for meta-programming because usually you can't go back to the original ET, especially if you're rewriting parts of the tree.

As an example, without the visitor mechanism, Extension node kinds inside of Quote (UnaryExpressions with node kind Quote) would be problematic. The Extensions would be black boxes, and Quote would be unable to substitute for ParameterExpressions in the black box. The

Quote mechanism would need to fully reduce all nodes to substitute the ParameterExpressions. Then the quoted expression would not have the shape or structure that is expected when using Quote. This would make meta-programming with such expression work poorly.

The ExpressionVisitor class is abstract with two main entry points and many methods for sub classes to override. The entry points visit an arbitrary Expression or collection of Expressions. The methods for sub classes to override correspond to the node types. For example, if you only care to inspect or act on BinaryExpressions and ParameterExpressions, you'd override VisitBinary and VisitParameter. The methods you override all have default implementations that just visit their children. If the result of visiting a child produces a new node, then the default implementations construct a new node of the same kind, filling in the new children. If the child comes back identity equal, then the default just returns it.

As an Extension node kind author, you should override Expression.VisitChildren to visit your sub expressions. Furthermore, if any come back as new objects, you should reconstruct your node type with the new children, returning the new node as your result. By default VisitChildren reduces the expression to a common node and then calls the visitor on the result. As discussed above, this is not the best result for meta-programming purposes, so it is important that Extension nodes override this behavior.

### 5.1.11 Annotations (Debug/Source Location Only)

We had a general annotation mechanism for a long time, but it kept presenting issues both in terms of the nature of the annotations and rewriting trees while correctly preserving annotations. We concluded that most annotations are short-lived for a single phase of processing, and they did not need node identity across tree rewrites. One common case of a persisted annotation, source locations, needed to span phases and rewrites, so we kept support for them specifically. We cut the general support for this release and will look at adding it back in a future release.

Note, all factory methods return new objects each time you call them. They return fresh objects so that you can associate them with unique annotations when that's needed. If you need caching for working set pressure or other performance turning (for example, re-using all Expression.Constant(1) nodes), then you need to provide that.

#### 5.1.11.1 Source Location Information

We model source location information with a DebugInfoExpression that represents a point in the ET where there is debugging information (a la .NET sequence points). A later instance of this class with the IsClear property set to True clears the debugging information. This node type has properties for start and end location information. It also can point to SymbolDocumentInfo which contains file, document type, language, etc.

## 5.2 LanguageContexts

LanguageContexts are the objects that represent a language that is implemented on the DLR and support the DLR's hosting model. These are the workhorse with many members that support various higher-level features in the DLR. The common hosting API is a clean layer on top of language implementation API and the LanguageContext. You can get call site binders from a

LanguageContext for creating DynamicExpressions in an expression tree so that the dynamic expression has that language's semantics for binding the operation at run time.

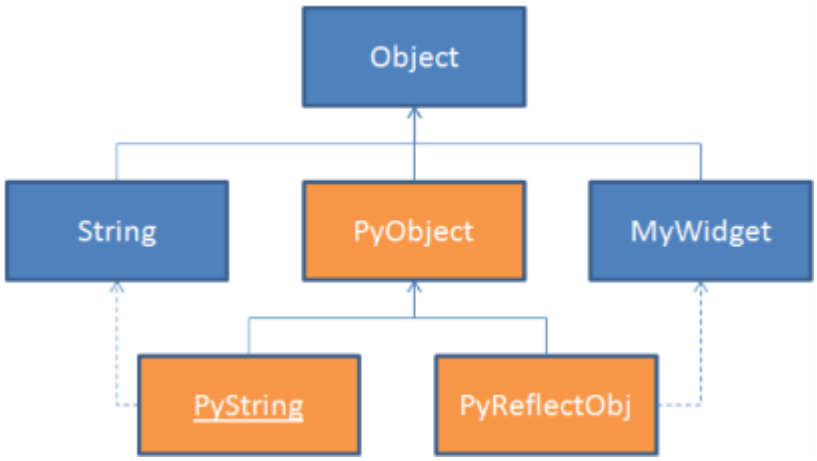
These have not been fully designed. The represent a mixture of some thought out architecture but also some haphazard collection of useful functionality. As we move to DLR v2, beyond CLR 4.0, we will clean these up and solidify the other parts of the language implementation API.

## 5.3 Type System

A key concept in the DLR is using .NET's Object as the root of the type system. Along with IDyanmicObject, these let dynamic languages easily and naturally talk to each other and share code. Equally important, dynamic languages should work well with existing powerful static languages on the platform such as VB.NET and C#. Their libraries should work well from dynamic languages.

### 5.3.1 Problems with Wrappers

Often implementations achieve interoperability through wrappers and marshaling layers, as this picture of Jython's system:




In this pattern the Python types exist in their own little world. For every underlying type there is a Python-specific wrapper. This standard pattern is okay for supporting a single language. As long as all your code is Python code all your objects are PyObjects, and they work great together with the Python-specific information on them. Where this pattern breaks down is when you want to integrate multiple languages. Then every time an object moves from one language to another it needs to be unwrapped from the source language and rewrapped appropriately for the destination. This can have performance issues as these wrapper objects are created and discarded for any cross-language calls.

The wrapper approach can have deeper problems. One challenge is just to figure out what object to pass. For example, if Python has a PyString, and it calls a C# function that expects an Object, should Python pass the PyString or should it unwrap it into a String? These kinds of subtle type issues never have a good answer. There are also big issues with losing object identity when objects are silently wrapped and unwrapped behind the programmer's back.

This wrapper pattern is used by many popular dynamic languages implemented in C. When implementing a dynamic language in C, these kinds of wrappers make a lot of sense because a C pointer doesn't have any useful runtime type information, so you need to decorate it with a layer that can provide the runtime type information that's needed. However, managed runtimes like the CLR provide rich type information for their standard objects, so it should be possible to use those objects directly without the confusion and cost of wrappers and marshalling. This is what the DLR uses for its type system.

### 5.3.2 Object and IDynamicMetaObjectProvider

The core of the DLR's type system is based on passing messages to objects. This is a consistent model in object-oriented systems. This simple notion doesn't explicitly talk about types at all, but instead focuses on objects and messages. Every dynamic and static language has its own notion of what a type is - from C#'s static single-inheritance types to Python's dynamic multiple-inheritance types to JavaScript's prototypes. Trying to reconcile all of these different systems at the type level is extremely complicated. Still, if you view these languages from an objects and messages perspective, there's a big commonality.

In order to support a broad range of languages in this kind of a type system, the DLR needs a standard set of messages. The set needs to be rich enough to capture an appropriately complete set of features in various languages while remaining sufficiently common so that code written in different languages can work together. The DLR has a good set of operations derived from working with several dynamic languages as well as C# and VB.NET. To see our current set of operations, see the discussion of DynamicObject or DynamicMetaObject.

Given this set of messages, the DLR needs some mechanisms to allow objects of different types to respond to these messages. While the DLR lets developers treat objects uniformly whether they come from a static or a dynamic language, under the hood the DLR needs to use different mechanisms to implement this message passing for the two different cases. If the type of the object is defined by a statically typed language, the DLR in conjunction with languages uses .NET reflection and standard CLS operator methods. The DLR also handles tasks like making delegates respond to the Invoke message. For types that are defined by a dynamic language, the DLR provides a more dynamic way to respond these messages. Types from dynamic languages implement IDynamicMetaObjectProvider, which lets them proffer custom DynamicMetaObjects for responding to messages at runtime. DynamicMetaObjects enable languages to build their custom behaviors into their objects so that they behave appropriately wherever they are used.

Each language avoids intrinsic knowledge about the details of the type system in the other languages. Languages just have to support the IDynamicMetaObjectProvider / DynamicMetaObject interoperability protocol and use it when working with objects from other languages.

## 5.4 IDynamicMetaObjectProvider and DynamicMetaObject

With just the dynamic call sites and binders logic defined above, you can already imagine implementing a language which takes advantage of the DLR’s caching system. You could emit a dynamic call site for each dynamic operation and write a binder that produces rules based on the types of the operands provided. Your binder could understand dynamic dispatch to static

.NET types (whose static structure can be gleaned from reflection), as well as dispatch to its own dynamic types, which it natively understands.

However, what if you want the binder for your language to dispatch to functions on objects created by another dynamic language? In this case, reflection will not help you as it would only show you the static implementation details behind that type's dynamic façade. Even if the object were to offer a list of possible operations to perform, there are still two problems. You would have to rebuild your language every time some new language or object showed up with a new operation your language didn't know about. You would also really like to perform these operations with the semantics of the target object's language, not your language, to ensure that object models relying on quirks of that language's semantics still work as designed.

Also, a library author might have a very dynamic problem domain they want to model. For example, if you have a library that supports drilling into XML data, you might prefer that your users' code to look like Customers.Address.ZipCode rather than XElement.Element("address").Element ("zipcode"), with the library allowing dynamic access to its data. Another example would be a library doing something similar with JSON objects returned from web services, or other highly dynamic data sources.

What's needed is a mechanism through which objects defined in various dynamic languages can offer to *bind their own operations* in the way that they see fit, while still obtaining the benefits of the DLR's caching infrastructure. The most visible part of this mechanism is the IDynamicMetaObjectProvider interface, which classes implement to offer dynamic dispatch support to a consuming language. The DynamicMetaObject class complements IDynamicMetaObjectProvider and serves as the common representation of dynamic metadata, allowing operations to be dispatched on dynamic objects.

### 5.4.1 IDynamicMetaObjectProvider

An object that offers up either its source language's semantics or its own type's custom dispatch semantics during dynamic dispatch must implement **IDynamicMetaObjectProvider**. The IDynamicMetaObjectProvider interface has a single method, GetMetaObject, which returns a DynamicMetaObject that represents this specific object's binding logic. A key strategy for DLR languages is to use .NET's Object as the root of their type hierarchy, and to use regular .NET objects when possible (such as IronPython using .NET strings). However, the IDynamicMetaObjectProvider protocol may still be needed for these base types at times, such as with IronRuby's mutable strings.

Library authors (even those using static languages) might implement IDynamicMetaObjectProvider so that their objects can present a dynamic façade in addition to their static interface. These objects can then present a better programming experience to dynamic languages as well as enable lighterweight syntax in languages such as C# with the 'dynamic' keyword.

### 5.4.2 DynamicMetaObject

An instance of **DynamicMetaObject** represents the binding logic for a given object, as well as the result for a given expression that's been bound. It has methods that allow you to continue composing operations to bind more complex expressions.

The three major components of a DynamicMetaObject are:

- The value, the underlying object or the result of an expression if it has one, along with information on its type.
- An expression tree, which represents the result of binding thus far.
- A set of restrictions, which represent the requirements gathered along the way for when this expression tree can serve as a valid implementation.

The expression tree and set of restrictions should feel familiar as they are similar in purpose to the implementation and test within a rule. In fact, when a DynamicMetaObject-aware binder such as DynamicMetaObjectBinder is asked to bind a given DynamicMetaObject, it uses the expression tree as the implementation and transforms the restrictions into the test for the rule.

Restrictions should be mostly of a static or invariant nature. For example, they test whether an object is of a specific static type or is a specific instance, something that will not change due to side effects during evaluation of this expression. The expression tree produced by the DynamicMetaObject binding may supply other tests that vary on dynamic state. For example, for objects whose member list itself is mutable, the expression tree may contain a test of an exact version number of the type. If this dynamic version test fails, the rule may be outdated, and the call site needs to call the binder and update the cache.

DynamicMetaObjects may also be composed to support merging several operations or compound dynamic expressions into a single call site. For example, `a.b.c.d()` would otherwise compile to three or four distinct dynamic call sites, calling three or four Target delegates each time the operation is invoked. Instead, the compiler may generate a single call site that represents the compound operation. Then, for each pass through this call site at runtime, the binder can decide how much of this compound expression it can bind in advance, knowing just the type of the source, `a`. If `a` has a certain static type, the return type of `a.b` may then be known, as well as `a.b.c` and the invocation `a.b.c.d()`, even before the methods are executed. In this case, the binder can return a single rule whose implementation covers the entire invocation.

### 5.4.3 DynamicMetaObjectBinder

If a language wants to participate not just in the DLR's caching system, but also interop fully with other dynamic languages, its binders need to derive from one of the subclasses of **DynamicMetaObjectBinder**.

DynamicMetaObjectBinder acts as a standard interoperability binder. Its subclasses represent various standard operations shared between most languages, such as GetMemberBinder and InvokeMemberBinder. These binders encode the standard static information expected across languages by these operations (such as the name of a method to invoke for InvokeMemberBinder). At runtime, DynamicMetaObject's Bind method accepts a target DynamicMetaObject and an array of argument objects. This Bind method, however, does not perform the binding itself. Instead, it first generates a DynamicMetaObject for each operand or argument as follows:

- **For objects that implement IDynamicMetaObjectProvider:** The Bind method calls .GetMetaObject() to get a custom DynamicMetaObject.
- **For all other static .NET objects:** The DLR generates a simple DynamicMetaObject that falls back to the semantics of the language, as represented by this binder class.

This subclass of DynamicMetaObjectBinder then calls out to the relevant Bind… method on the target DynamicMetaObject (such as BindInvokeMember), passing it the argument DynamicMetaObjects. By convention all binders defer to the DynamicMetaObjects before imposing their language's binding logic on the operation. In many cases, the DynamicMetaObject is owned by the language, or if it is a default DLR .NET DynamicMetaObject, it falls back to the language binder for deciding how to bind to static .NET objects. Deferring to the DynamicMetaObject first is important for interoperability.

Binding using DynamicMetaObjectBinders takes place at a higher level by using DynamicMetaObjects instead of the .NET objects themselves. This lets the designer of dynamic argument objects or operands retain control of how their operations are dispatched. Each object's own defined semantics always take highest precedence, regardless of the language in which the object is used, and what other objects are used alongside them. Also, because all languages derive from the same common binder classes, the L0 target methods that are generated can cache implementations across the various dynamic languages and libraries where objects are defined, enabling high performance with semantics that are true to the source.

#### 5.4.3.1 Fallback Methods – Implementing the Language's Semantics

A major principle in the DLR design is that "the object is king". This is why DynamicMetaObjectBinder always delegates binding first to the target's DynamicMetaObject, which can dispatch the operation with the semantics it desires, often those of the source language that object was defined in. This helps make interacting with dynamic object models feel as natural from other languages as it is from the language the object model was designed for.

However, there are often times when you'll write code that uses a language feature available in your language, but not in the language of your target object. For example, let's say you're coding in Python, which provides an implicit member __class__ on all objects that returns the object's type. When dynamically dispatching member accesses in Python, you'll want the __class__ member to be available not just on Python objects, but also on standard .NET objects, as well as objects defined in other dynamic languages. This is where Fallback methods come in.

Each of the DynamicMetaObjectBinder subclasses defines a **fallback method** for the specific operation it represents. This fallback method implements the language's own semantics for the given operation, to be used when the object is unable to bind such an operation itself. While a language's fallback methods may supply whatever semantics the language chooses, the intention is that the language exposes a reasonable approximation of its compile-time semantics at runtime, applied to the actual runtime types of the objects encountered.

For example, the SetMemberBinder class defines an abstract FallbackSetMember method that languages override to implement their dynamic member assignment semantics. At a member assignment call site, if the target DynamicMetaObject's BindSetMember method can't dispatch the assignment itself, it can delegate back to the language's SetMemberBinder. The DynamicMetaObject does this by calling the binder's FallbackSetMember method, passing the target DynamicMetaObject (itself) and the value to assign to the member.

There are specific subclasses of DynamicMetaObjectBinder for each of 12 common language features in the interoperability protocol. For each feature you wish to support binding in your language, you must implement a binder that derives from its class. The API Reference section

blow describes these operation binder abstract classes and provides more details on implementing them.

#### 5.4.3.2 Error Suggestions for Dynamic Binding when Static Binding Fails

An MO that wants even finer control over the "fallback dance" may also choose to pass an **error suggestion** to a language's fallback method. An error suggestion supplies a binding recommendation to the language binder, which it is encouraged to use if it fails the binding process. The standard use for this mechanism is to supply an MO containing an expression that binds to dynamic members of the target object. The language gets to do static member binding first, but it should then honor this suggestion if it is non-null rather than throw an error.

Preferring language-based static member binding over the dynamic member binding is used by the DynamicObject abstract class. This is described later in this document.

#### 5.4.3.3 DynamicMetaObjectBinder Subclasses

There are specific subclasses of DynamicMetaObjectBinder for each of 16 common language features in the interoperability protocol. For each feature you wish to support binding in your language, you must implement a binder that derives from its class. The interoperability subclasses are:

| | |
|---|---|
| **GetMemberBinder** | Represents an access to an object's member that retrieves the value. In some languages the value may be a first-class function, such as a delegate, that closes over the instance, o, which can later be invoked.<br>*Example:* o.m<br>If the member doesn't exist, the binder may return an expression that creates a new member with some language-specific default value, returns a sentinel value like $Undefined, throws an exception, etc. |
| **SetMemberBinder** | Represents an access to an object's member that assigns a value.<br>*Example:* o.m = 12<br>If the member doesn't exist, the binder may return an expression that creates a new member to hold this value, throws an exception, etc. |
| **DeleteMemberBinder** | Represents an access to an object's member that deletes the member.<br>*Example:* delete o.m<br>This may not be supported on all objects. |
| **GetIndexBinder** | Represents an access to an indexed element of an object that retrieves the value.<br>*Example:* o[10] *or* o["key"]<br>If the element doesn't exist, the binder may return an expression |

| | |
|---|---|
| | that creates a new element with some language-specific default value, return a sentinel value like $Undefined, throw an exception, etc. |
| **SetIndexBinder** | Represents an access to an indexed element of an object that assigns a value.<br><br>*Example:* o[10] = 12 *or* o[“key”] = value<br><br>If the member doesn't exist, the binder may return an expression that creates a new element to hold this value, throw an exception, etc. |
| **DeleteIndexBinder** | Represents an access to an indexed element of an object that deletes the element.<br><br>*Example:* delete o.m[10] *or* delete o[“key”]<br><br>This may not be supported on all indexable objects. |
| **InvokeBinder** | Represents invocation of an invocable object, such as a delegate or first-class function object.<br><br>*Example:* a(3) |
| **InvokeMemberBinder** | Represents invocation of an invocable member on an object, such as a method.<br><br>*Example:* a.b(3)<br><br>If invoking a member is an atomic operation in a language, its compiler can choose to generate sites using InvokeMemberBinder instead of GetMemberBinder nested within InvokeBinder. For example, in C#, a.b may be a method group representing multiple overloads and would have no intermediate object representation for a GetMemberBinder to return.<br><br>However, a language that chooses to emit InvokeMemberBinders might try a method invocation on any dynamic object. To support all DLR languages, objects from dynamic libraries must support both Invoke and InvokeMember operations. However, to generalize the implementation of InvokeMember, a dynamic object may fall back to the Invoke functionality in the language binder. Languages that define an InvokeMemberBinder are therefore required to implement the FallbackInvoke method alongside FallbackInvokeMember. Dynamic objects can then implement InvokeMember by resolving a GetMember operation themselves using their own semantics, and passing the resulting target DynamicMetaObject on to FallbackInvoke, which can be implemented by delegating to the language’s existing InvokeBinder functionality. |
| **CreateInstanceBinder** | Represents instantiation of an object with a set of constructor arguments. The object represents a type, prototype function, or |

| | |
|---|---|
| | other language construct that supports instantiation.<br><br>*Example:* new X(3, 4, 5) |
| **ConvertBinder** | Represents a conversion of an object to a target type.<br><br>This conversion may be marked as being an implicit compiler-inferred conversion, or an explicit conversion specified by the developer.<br><br>*Example:* (TargetType)o |
| **UnaryOperationBinder**<br><br>**BinaryOperationBinder** | Represents a miscellaneous operation, such as a unary operator or binary operator, respectively.<br><br>*Examples:* a + b, a * b, -a<br><br>Contains an Operation string that specifies the operation to perform, such as Add, Subtract, Negate, etc.. There is a core set of operations defined that all language binders should support if they map reasonably to concepts in the language. Languages may also define their own Operation strings for features unique to their language, and may agree independently to share these strings to enable interop for these features.<br><br>The operator strings in the core set are:<br><br>Decrement, Increment, Negate, Positive, Not, Add, Subtract, Multiply, Divide, Mod, ExclusiveOr, BitwiseAnd, BitwiseOr, LeftShift, RightShift, Equals, GreaterThan, LessThan, NotEquals, GreaterThanOrEqual, Power, LessThanOrEqual, InPlaceMultiply, InPlaceSubtract, InPlaceExclusiveOr, InPlaceLeftShift, InPlaceRightShift, InPlaceMod, InPlaceAdd, InPlaceBitwiseAnd, InPlaceBitwiseOr, InPlaceDivide, InPlacePower |