IMPROVING END-USER CONTROL THROUGH DLR INTEGRATION WITH SILVERLIGHT

Author
Manish Agrawal
# CONTENTS

Introduction 4

Why DLR? 5

- Dynamic Type System 5
- Dynamic Code Generation 5
- Hosting API 6
  - Adding Dynamic Attribute Support to C++ CLR Type 6

Sample Application 8

- Silverlight DLR Integration 11

Adding Dynamic Attributes to C# Types 11

Conclusion 14
INTRODUCTION

The IronPython DLR engine and .NET DLR (dynamic language runtime) can be combined with the rich UI capabilities of Silverlight to create an application that provides the end user with more control than ever before.

This paper illustrates the concept using a capital markets blotter as a sample implementation. However, the concepts presented here are equally relevant to applications across any vertical that share one or more of the following characteristics:

- Ability to frequently access changing data in real time
- Ability to allow data to be visualized in different perspectives (grid, text, bar and line charts, etc.)
- Ability to perform dynamic computations on the data and use those to make time-critical decisions

Some of the commonly used technical solutions for the above requirements include but are not limited to:

- Creating an application that can export current data into a spreadsheet application like Excel and then performing the computations in Excel.
- Building the application as several Excel macros that connect to web services to obtain data in Excel where a user can then perform the run time computations.
- Providing various predecided combinations of calculations as “canned” options in a Windows or web application.

However, all fall short of what is really required — to provide a user with a macro language, along with a domain specific language (DSL), that can be applied dynamically to live streaming data (and not just snapshots) to perform computations. The DSL will consist of a pre-packaged library of custom functions created for analysis that may be specific to an organization or across a particular vertical.
THE BASIC PREMISE BEHIND THE NEED FOR DLR IS TO ALLOW THE USER TO USE BASE DATA (PRESENTED BY THE APPLICATION) FOR PERFORMING CALCULATIONS DYNAMICALLY AND VIEWING THE RESULTS IMMEDIATELY—WHETHER IT IS BY WAY OF A NEW COLUMN IN A DATAGRID OR A NEW GRAPH (SERIES) IN A CHART.

DYNAMIC LANGUAGE RUNTIME (DLR) IS A SET OF SERVICES WRITTEN ON TOP OF THE COMMON LANGUAGE RUNTIME (CLR). THE DLR ALLOWS DYNAMIC LANGUAGES LIKE PYTHON AND RUBY TO BE USED AS .NET LANGUAGES TO DEVELOP WHOLE APPLICATIONS OR SPECIFIC PARTS OF AN APPLICATION THAT ARE MORE DIFFICULT TO BUILD USING STATICALLY TYPED LANGUAGES LIKE C#.

SOME KEY FEATURES OF THE DLR INCLUDE:

**DYNAMIC TYPE SYSTEM**

In a dynamic language like Python, a variable is assumed to have a type based on the value assigned to it. For instance, in the Python expression X=24, X is considered as an integer. In addition, attributes in a class do not need to be declared and can simply be used.

```python
class Person:
    def SayHello(self):
        Print 'hello ' + self.FirstName
    #FirstName is not declared anywhere.
X=Person()
x.SayHello() #this results in an error that the Attribute FirstName is missing.
x.FirstName='MyName'
x.SayHello() #this prints 'hello MyName'
```

This ability to dynamically add attributes and methods to an existing class is useful in applications that need to support dynamic computations.

**DYNAMIC CODE GENERATION**

Hosting a DLR in a regular C# application allows code snippets, like the one shown in the example below, to be injected as compiled code at run time. The following is a plain C# console application:

```csharp
using System;
using IronPython.Hosting;
using Microsoft.Scripting.Hosting;
namespace ConsoleApplication1
{
    class Program
    {
        static void Main(string[] args)
        {
            string definition = @"def SayHello(x): print 'hello ' + x.FirstName";
            ScriptEngine engine = Python.CreateEngine();
            ScriptSource source = engine.CreateScriptSourceFromString(definition);
            ScriptScope scope = engine.CreateScope();
            source.Execute(scope);
        }
    }
}
```
In the above code, the “definition” variable is a simple string. This string value is converted into executable Python code at runtime.

Hosting API

When hosting a DLR in a C# application, a reference to a Python function can be obtained as a regular C# delegate and executed with C# objects passed in as parameters. The following listing completes the above example:

```csharp
using System;
using IronPython.Hosting;
using Microsoft.Scripting.Hosting;
namespace ConsoleApplication1
{
    public class Person
    {
        public string FirstName { get; set; }
        public string LastName { get; set; }
    }

class Program
{
    static void Main(string[] args)
    {
        string definition = @"def SayHello(x): print 'hello ' + x.FirstName";
        ScriptEngine engine = Python.CreateEngine();
        ScriptSource source = engine.CreateScriptSourceFromstring(definition);
        ScriptScope scope = engine.CreateScope();
        scope.Execute(source);
        Person x = new Person { FirstName = "manish" ];
        Action<Person> SayHelloToPerson = 
            scope.GetVariable<Action<Person>>("SayHello");
        SayHelloToPerson(x);
        Console.ReadLine();
    }
}
```

The above code converts a string variable “definition” into executable Python code at run time, gets a reference to the function created in Python into a C# delegate “SayHelloToPerson” and, finally, calls this delegate passing in a C# object of type “Person”. The Python code is able to operate on the C# Person object. However, the code cannot treat the Person object as a dynamic object (as described in the Dynamic Type System section above). To do that, some additional steps are required as described below.

Adding Dynamic Attribute Support to C# CLR Type

While an IronPython type will allow dynamic attributes to be added to it (via DLR) out of the box, a C# type does not provide this type of built-in support. To allow the DLR to add attributes dynamically to a C# type (for instance, a Person type), a few “special” methods need to be exposed on the C# type. When the DLR is asked to set or get an attribute on a C# type, it will invoke these special methods if the attribute is not found in the type’s “CLR definition”. The DLR supports this invocation at multiple points in the life cycle. For instance, when attempting a fetch, the DLR provides the ability to expose a method that will be invoked prior to the CLR “definition check”.
as well as after the check. Similarly, when attempting to set a value on the object, the DLR allows us to expose methods that will be invoked before and after the check. This can be seen in the following code snippet:

```csharp
public partial class Person
    Dictionary<string, object> customAttributes;

    [SpecialName]
    public void SetMemberAfter(string memberName, object value)
        {
            customAttributes.Add(memberName, value);
        }
    [SpecialName]
    public object GetBoundMember(string name)
        {
            return (customAttributes.ContainsKey(name) ? customAttributes[name] : null);
        }

This class uses a dictionary to store any dynamic attributes that are assigned to it. Method SetMemberAfter is invoked by the DLR when the Python code attempts to set a dynamic attribute on a Person object. Method GetBoundMember is invoked when we use the DLR to fetch the value of the dynamic attribute from the Person object. The SpecialName attribute is used for the compiler [and is not specific to the DLR].

_Note —_ With the introduction of dynamic types in .NET 4.0, C# types can extend from the DynamicObject class and override the TryGetMember and TrySetMember methods to achieve the same effect as above. If using the DynamicObject as a base class is not an option, the other alternative is to implement the interface IDynamicMetaObjectProvider.
SAMPLE APPLICATION

The sample Silverlight blotter application used here is a simplified version of an Order Management System with several features.

The application contains a DataGrid that displays a “real-time” view of order data in the system where the grid is frequently refreshed with both updated and new orders “streamed” to it from a WCF “order simulation” service.

The application also contains a moveable/resizable window to display “On demand” stock charts that showed a typical “Price versus Time” line graph as well as a “Volume versus Time” area graph. The on demand aspect alludes to the fact that the window tracks the user’s current item selection on the order DataGrid and updates the chart to reflect data corresponding to the order’s stock symbol. The chart plots graphs using data that is pushed to it asynchronously by a WCF “quote simulation” service.
The application provides a window that allows a user to add a new attribute to Order or Stock Quote data by naming the attribute and assigning an expression (formula) to it. The key here is that the user types an expression as a string. This string, in turn, represents the code that should be executed to compute a value for that attribute. For instance, if an Order item contains design time attributes, such as “ExecutedQuantity” and “Quote”, we need to allow the user to create a new attribute definition, such as “order.ExecutedValue = order.ExecutedQuantity * order.Quote.LastTradedPrice”. This feature causes a new attribute to be added to all objects of the specific type in the collection [used as a data source for the grid or chart]. When adding a new attribute to the Order type, a new column is also added to the DataGrid and displayed immediately with the computed value of the attribute for each row. When a new attribute is added from the “Chart window,” the attribute is added to each StockQuote item in a collection and the values thus computed are used to plot an additional line series on the same chart that displays the “Price versus Time” graph. Changing the order item selection on the DataGrid causes all the graphs (including the ones added dynamically) to be refreshed with the new selection’s stock quote symbol data.
To aid the user in dynamic attribute creation, the application provides a special “IntelliSense” or “Auto-complete” feature that displays a list of existing attributes on an order or stock quote as well as additional macro-like functions that are made available via a separate IronPython function library (e.g. an analytics or statistics library). For instance, typing in “order.” causes a dropdown to appear that lists all “design time” attributes of an “order object” as well as any other attributes that have been dynamically added at any time up until the dropdown is displayed.
This section uses the methods of integrating DLR in a .NET application explained above to implement the features required in the sample application.

Adding Dynamic Attributes to C# Types

The IronPython engine (version 2.6 RC1 for .NET 2.0) is used to add dynamic attributes to the C# Order/StockQuote class. There are several steps required to integrate with the DLR in this scenario:

- **IronPython Setup** — This involves initializing the engine and creating an initial string-based definition of an IronPython function that will accept an object as a parameter and manipulate the object. In our case, it will be used to add dynamic attributes to a C# object (Order/StockQuote) passed to the function.

```csharp
public class PythonEngine
{
    public PythonEngine()
    {
        ScriptRuntime runtime = new ScriptRuntime(setup);
        ScriptEngine pyEngine = runtime.GetEngine("python");
        ScriptScope scope = pyEngine.CreateScope();
        //IronPython update function
        codeBuilder = new StringBuilder();
        codeBuilder.AppendLine("def updateItem(order):");
        codeBuilder.AppendLine("    pass");
    }
}
```

The Silverlight framework provides its own version of the ScriptRunTime setup where it ensures that all required assemblies are already loaded by the PythonEngine. A ScriptScope is similar to a namespace within which functions and variables can be defined. In the code above, a StringBuilder is used to store the initial definition of the function that will be applied to each item in the Order and StockQuote collection. Note that “pass” is the Python equivalent of a no-op.

Separate instances of the custom “PythonEngine” class (a wrapper around the PythonEngine) are used to handle each collection; each instance, therefore, stores the latest function definition for an Order or StockQuote type. We use a StringBuilder to store the function definition, since it can then be appended with all the dynamic attribute definitions that a user provides. It is important to note that at this stage [setup], we only have a string representation and not the actual IronPython function.

- **Defining Dynamic Attributes** — This is where the user provides an attribute definition and expects that the attribute will be added to (and computed for) each item of the specified type and any other items that are added to the associated collection. For instance, this is the sequence of events that occur when the user specifies a new attribute: “order.TotalExecuted = order.OrderedQuantity — order. LeavesQuantity”:

Method “UpdateItemDefinition” on PythonEngine instance is invoked — this method is responsible for adding the new attribute string definition to the StringBuilder. In effect, the StringBuilder should now contain the string:
def updateItem(order):
    pass
order.TotalExecuted = order.OrderedQuantity - order.LeavesQuantity
public class PythonEngine
{
    public void UpdateItemDefinition(string definition)
    {
        codeBuilder.AppendFormat("{0}{1}{2}", tab, definition, Environment.NewLine);
    }
}

Note that a tab is inserted before each line in accordance with Python syntax.

- **Converting the String to Code** — Once the string definition of the function is updated, it is now time to create/update the IronPython function representation of this string. This demonstrates the true value of IronPython in our application, providing the ability to take a string and treat it as code, as well as to add attributes dynamically to C# (and IronPython) data types. The idea is to convert the string into an IronPython function and then obtain a C# based “Action” delegate to that function.

```csharp
public Action<Order> UpdateItem = default(Action<Order>); // this is the C# delegate to the IronPython function
protected void LoadScript()
{
    ScriptSource codeScriptSource = pythonEngine.CreateScriptSourceFromString
    (codeBuilder.ToString(), Microsoft.Scripting.SourceCodeKind.Statements);
    CompiledCode compiledCode = codeScriptSource.Compile();
    compiledCode.Execute(scope);
    // get function delegate
    UpdateItem = scope.GetVariable<Action<Order>>("updateItem");
}
```

The first line asks the Python DLR engine to treat the string passed to it as IronPython source code. The second line creates a compiled version of that code (this is optional, but recommended for performance reasons). Finally, the code is executed and a reference to the “updateItem” IronPython function is obtained via the “GetVariable<T>” method exposed by the DLR.

- **Binding Silverlight Control to Dynamic Attribute** — While we have explored what it takes to update an IronPython function definition to add dynamic attributes to an Order object, we have not yet applied this function to items in the Order collection or added a new column to the DataGrid for the new attribute. While adding a new column to a DataGrid is easy and straightforward, binding it to a dynamic attribute of its item source requires additional work since Silverlight 3.5 does not have the ability to auto detect dynamic attributes added to a C# type. To make it work, a ValueConverter is used. To begin with, a new column is added to the DataGrid using the code snippet below.
DataGridTextColumn newColumn = new DataGridTextColumn();
Binding dynBinding = new Binding();
dynBinding.Converter = new PythonFieldConverter(pythonEngine);
dynBinding.ConverterParameter = dynamicAttributeName;
newColumn.Binding = dynBinding;

"PythonFieldConverter" is a ValueConverter that is "DLR aware". The name of the dynamic attribute is passed to it as a converter parameter so that it knows which specific value to use for this particular column. We also pass it a reference to the instance of our custom "PythonEngine" class, which has been used to update the IronPython function definition and contains the delegate to the "updateItem" function. Here is the code for the converter:

```csharp
class PythonFieldConverter : IValueConverter
{
    public object Convert(object value, Type targetType, object parameter, System.Globalization.CultureInfo culture)
    {
        Order orderItem = value as Order;
        string attributeName = parameter.ToString();
        pythonEngine.UpdateItem(orderItem);
        pythonEngine.TryGetMember(value, attributeName, out attributeValue);
        return attributeValue;
    }
}
```

When invoked, the value converter passes the current instance of the Order item through the IronPython "updateItem" function, which adds all dynamic attributes included by the user to the item. The converter then attempts to fetch the value of the newly computed attribute by invoking the DLR engine’s "TryGetMember" method.
In this paper, we looked at how the features of dynamic language runtime can be integrated with a popular rich UI platform like Silverlight to provide a DSL-based macro language to an application. We used the ability of a dynamic language like Python to interpret string representations of code into compiled code at run time and make it available in the C# application. We also explained how to use this dynamic behavior to display new columns in a grid.

Note: To view a sample implementation in a complete application that demonstrates the various concepts explained in this whitepaper, visit http://www.sapient.com/content/dam/sapient/sapientglobalmarkets/pdf/thought-leadership/reference-application.zip.

Using DLR, an application can provide scripting features similar to those in Excel VBA. Applications that deal with frequently changing data, such as financial analysis applications or management information systems, can benefit most from such features by allowing users to create their own customizations like runtime filters and formulae.
About Sapient Global Markets
Sapient Global Markets, a division of Sapient® (NASDAQ: SAPE), is a leading provider of services to today’s evolving financial and commodity markets. We provide a full range of capabilities to help our clients grow and enhance their businesses, create robust and transparent infrastructure, manage operating costs, and foster innovation throughout their organizations. We offer services across Advisory, Analytics, Technology, and Process, as well as unique methodologies in program management, technology development, and process outsourcing. Sapient Global Markets operates in key financial and commodity centers worldwide, including Boston, Chicago, Houston, New York, Calgary, Toronto, London, Amsterdam, Düsseldorf, Geneva, Munich, Zurich, and Singapore, as well as in large technology development and operations outsourcing centers in Bangalore, Delhi, and Noida, India.

For more information, visit www.sapientglobalmarkets.com.

© 2012 Sapient Corporation.
Trademark Information: Sapient and the Sapient logo are trademarks or registered trademarks of Sapient Corporation or its subsidiaries in the U.S. and other countries. All other trade names are trademarks or registered trademarks of their respective holders.
Global Offices

Amsterdam
Prins Bernhardplein 200
1097 JB Amsterdam
Netherlands
Tel: +31 (0) 20 330 0011

Bangalore
Salarpuria GR Tech Park
6th Floor, "VAYU" Block
#137, Bengaluru 560066
Karnataka
India
Tel: +91 (080) 410 47 000

Boston
131 Dartmouth Street
3rd Floor
Boston, MA 02116
Tel: +1 (617) 621 0200

Calgary
888 3rd Street SW
Suite 1000
Calgary, Alberta T2P 5C5
Canada
Tel: +1 (403) 444 5574

Chicago
30 West Monroe,
12th floor
Chicago, IL 60603
Tel: +1 (312) 458 1800

Delhi
Towers D & E,
DLF Cyber Greens
DLF Phase III
Sector 25-A
Gurgaon 122002
Haryana
India
Tel: +91 (124) 416 7000

Düsseldorf
Speditionstrasse 21
40221 Düsseldorf
Germany
Tel: +49 (0) 211 540 34 0

Geneva
Succursale Genève
c/o Florence Thiébaud, avocate
rue du Cendrier 15
1201 Geneva
Switzerland
Tel: +41 (0) 58 206 06 00

Houston
Heritage Plaza
1111 Bagby Street Suite 1950
Houston, TX 77002
Tel: +1 (713) 493 6880

Los Angeles
1601 Cloverfield Blvd.
Suite 400 South
Santa Monica, CA 90404
Tel: +1 (310) 264 6900

London
Eden House
8 Spital Square
London, E1 6DU
United Kingdom
Tel: +44 (0) 207 786 4500

Munich
Arnulfstrasse 60
80335 München
Germany
Tel: +49 (0) 89 552 987 0

New York
40 Fulton Street
22nd Floor
New York, NY 10038
Tel: +1 (212) 206 1005

Singapore
Air View Building #02-01
2 Peck Seah Street
Singapore 079305
Republic of Singapore
Tel: +65 (3) 104 5308

Toronto
129 Spadina Avenue
Suite 500
Toronto, Ontario M5V 2L3
Canada
Tel: +1 (416) 645 1500

Zürich
Seefeldstrasse 35
8008 Zürich
Switzerland
Tel: +41 (58) 206 06 00