Dr.Dobb's
THE WORLD OF SOFTWARE DEVELOPMENT

Graphr: A Plugin-Based Graphical App in C# Using MEF and Embedded IronPython

Using .NET 4.0's MEF to facilitate a plugin architecture turns out to be fairly easy. So is embedding IronPython. Doing both has its challenges

June 03, 2011
URL:http://www.drdobbs.com/windows/graphr-a-plugin-based-graphical-app-in-c/229700040

Graphr is a Windows Presentation Foundation-based GUI program implemented in C# that allows you to generate a dataset based on a rule (expression) such as **"(x-3) * (x-5)"** for a given range of values. Graphr then displays a graph on the screen. Graphr is interesting because it is a polyglot application. It is a C# application that hosts an embedded IronPython engine. It allows you to write your rules in Python and even use any function from Python's math library, such as **sin()** or **cos()**. The best part is that you don't need to parse and evaluate the expression. Python will do this for you. The other interesting aspect of Graphr is its architecture. Graphr is a plugin-based system. It utilizes the Managed Extensibility Framework (MEF) to load graph plugins dynamically and use them in a totally decoupled fashion. You can add additional graphs and Graphr will be happy to automatically use them without touching Graphr itself. This is a useful practice that can be applied in many situations.

In this article, we will demonstrate Graphr, explain in detail how to embed IronPython in any application, and explore MEF and its use in a plugin architecture. In a separate article, which you can access immediately here, we've done a walkthrough of Graphr's architecture. Finally, we'll talk a little about the future of Graphr.

Graphr in Action

Before we delve into the code, let's have some fun playing with Graphr. You need Visual Studio 2010 and IronPython 2.7. Both are free (Visual Studio 2010 has non-free versions, too, but they are not needed to build Graphr).

When you launch Graphr, you see a window divided into two panes. The left pane contains the rule and range text boxes and a "Visualize" button on top, and a graph selector and graph properties on the bottom. The right pane (empty initially) contains the graph display area. Click the "Visualize" button and a yellow line graph is displayed (Figure 1).

[Click image to view at full size]

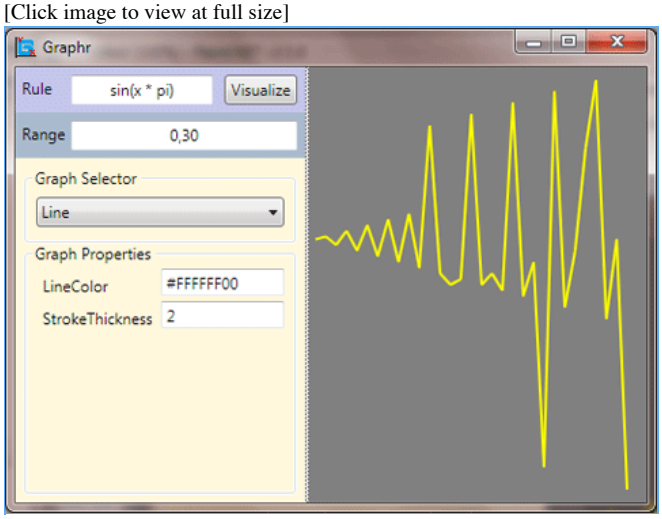

Figure 1.

The graph properties area allows you to customize the appearance of the graph. The LineColor text box contains four double-digit hex numbers (**A, R, G, B**). The first two digits control transparency (**00** fully transparent, **FF** fully opaque), the other digits control the red, green, and blue values of the color. The **LineThickness** text box controls the thickness of the line.

You can also change the rule and the range. When you change the rule and/or the range, you must click the "Visualize" button again. The graph selector dropdown box allows you to select a different graph type. Figure 2 shows a bar graph that uses a different rule and a range that includes negative numbers. Note that each graph type has its own graph properties. The bar graph uses a gradient brush to paint the bars, so the bar color changes gradually from the selected color at the top to white at the bottom of the bar.

[Click image to view at full size]

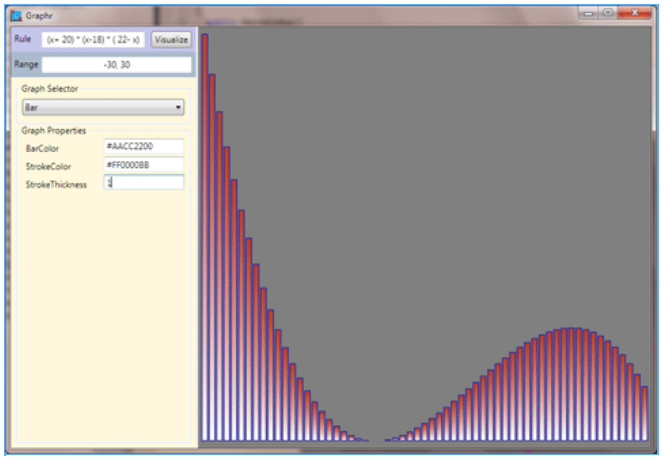

Figure 2.

Graphr is also fully resizable. You can resize the entire window and the graph will adjust and fit into the new size. You can also change the ratio between the left and right pane by dragging the splitter left and right.

Graphr also supports a pie graph, which displays slices that are proportionate to the values of the rule. The color of each slice is selected randomly, and you can only control the border color. If you play with the pie graph, it is recommended that you use a range of up to 50 values because it takes longer to render (especially if you resize the window).

Embedding IronPython

Graphr is able to parse and evaluate complicated expressions. Implementing a custom parser and evaluator in C# is possible, but would take a long time. Instead, we opted to take advantage of Python, which provides out of the box dynamic evaluation capability.

IronPython is an open source implementation of Python for the .NET platform. IronPython 2.7, which is what we used in Graphr can be downloaded here.

Dynamic Evaluation with IronPython

IronPython makes it easy to dynamically evaluate any valid Python expression and, in particular, mathematical expressions. The key is the built-in **eval()** function, which takes a Python expression as a string, executes it dynamically, and returns the resulting Python object. Here is a short interactive session that demonstrates it:

```
>>> eval('5 + 5')
10
>>> x = 3
>>> eval('x + 2')
5
>>> x = 'hello'
>>> eval('x + " world"')
'hello world'
>>>
```

Graphr needs to evaluate an expression that contains an x variable multiple times, where **x** is assigned different values from the range. The Evaluatr.py Python module provides this functionality. It imports all the functions in Python's math library so they can be evaluated as part of the expression. It contains two functions: **evaluateSingleValue()** and **evaluate()**. The **evaluateSingleValue()** function gets an expression (rule) and an **x** value. It substitutes all the **x**'s in the expression with the value of **x** and evaluates it, as shown here:

```
import os
from math import *

def evaluateSingleValue(expr, x):
  # replace the 'X' in the expression with the current value
  e = expr.replace("x", str(x))

  # evaluate the substituted expression string using the eval() function
  return eval(e)
```

The **evaluate()** function is used directly by Graphr. It accepts an expression and a list of values. It iterates over the values and evaluates each one using **evaluateSingleValue()**. Note that **evaluate()** is a Python generator, which means that values are computed only as they are needed. The **yield** keyword returns control to the caller, which allows efficient streaming in case of large ranges that don't need to be fully evaluated (such as a graph that displays a window of 100 values regardless of the entire range), as shown next:

```
def evaluate(expr, values):
   """Evaluate an expression with an X variable over a range of X values

   expr - a string that represents a Python expression with a variable
   values - a list of floating point values

   return a generator that yields pairs of (x, eval(expr))
```

```
    """
    expr = expr.lower()
    for x in values:
      try:
        yield (x, evaluateSingleValue(expr, x))
      except:
        yield (None, None)
```

Embedding IronPython in a C# Program

Now, that we have a good Python evaluator that can dynamically evaluate Python expressions, it's time to embed it in your C# program. IronPython is built on top of the DLR (Dynamic Language Runtime). The DLR exposes hosting APIs designed to support this use case. We isolated all embedding code into a single C# class called **IronPythonEvaluator**. The **IronPythonEvaluator** relies on the Evaluatr.py module to perform the actual computation. This raises the question of how to deploy the Evaluatr.py module with the Graphr app. The most straightforward approach is just to provide it as an additional file, but we chose a more integrated approach where the source code for the module is embedded as a resource in the Graphr.exe assembly (Figure 3).

[Click image to view at full size]

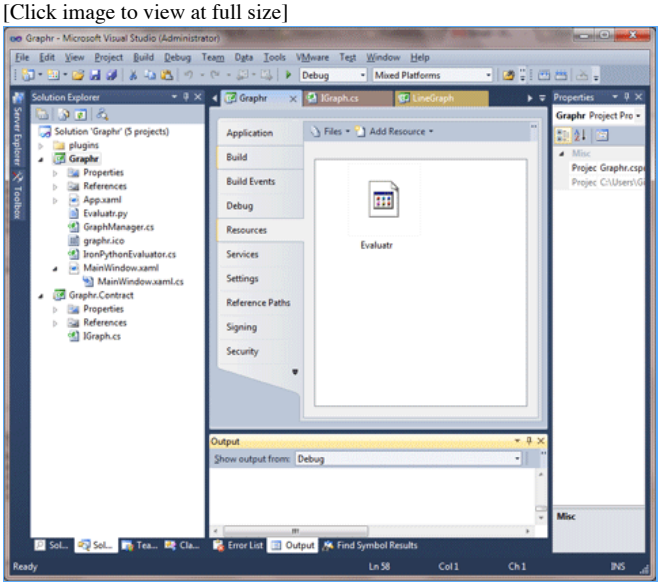

Figure 3.

When the **IronPythonEvaluator** is initialized, it loads the resource and saves it to a file. This file is used later by the IronPython runtime engine. Before we dive into the **IronPythonCalculator**, here are the assemblies it uses:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

// The following assemblies are needed to host IronPython
using IronPython.Hosting;
using Microsoft.Scripting.Hosting;

using Graphr.Properties;
using System.IO;
```

The **IronPythonEvaluator** has a constructor and two overloaded **evaluate()** methods. The constructor creates an IronPython engine with a "Debug" option. This is useful because it allows stepping from C# into IronPython code when debugging in VisualStudio. Also, the debugger will present you with a unified call stack from C# to Python if an exception is raised in the Python code (Figure 4). This polyglot debugging is a unique feature of the .NET platform.

[Click image to view at full size]

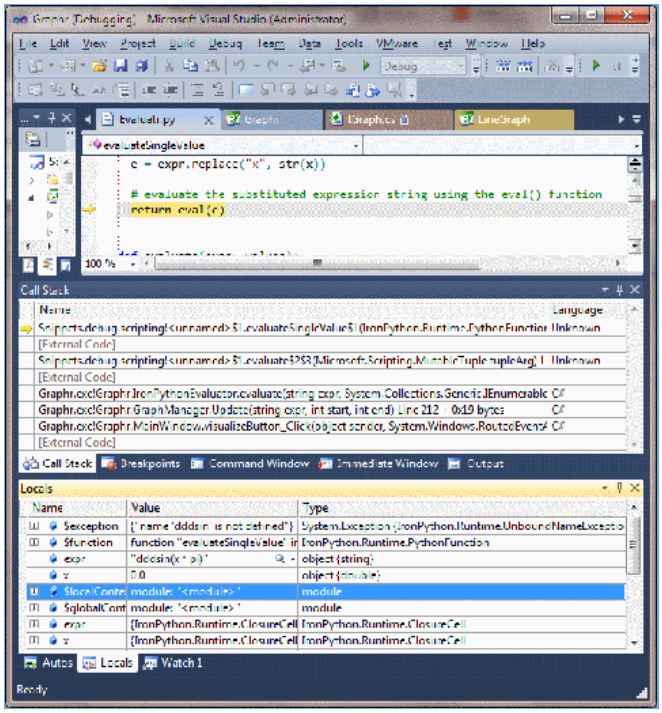

Figure 4.

The constructor then adds the IronPython Lib directory to the engine's search path. That allows IronPython to import any standard library module. The next step is to write the embedded Evaluatr.py module to a temporary file, then load this file into the engine's runtime and assign the resulting Python module into the **evaluatr** dynamic member. Once the file has been loaded into the runtime, it can be deleted. Note, that this code is not industrial strength and doesn't handle exceptions so the temporary file will not be deleted if an exception is thrown.

```
namespace Graphr
{
  class IronPythonEvaluator
  {
    public IronPythonEvaluator()
    {
      //Creating a new script runtime
      Dictionary options = new Dictionary();
      options["Debug"] = true;
      var env = Python.CreateEngine(options);

      // Set the default search path for IronPython (not set by default)
      var searchPaths = env.GetSearchPaths();

      // Note: this is hard-coded path to the IronPython 2.7 lib directory
      var p = "C:/Program Files (x86)/IronPython 2.7/Lib";
      if (!Directory.Exists(p))
      {
          p = "C:/Program Files/IronPython 2.6 for .NET 4.0/Lib";
      }

      searchPaths.Add(p);
      env.SetSearchPaths(searchPaths);
      var filename =  Path.GetTempPath() + "Evaluatr.py";
      // Write the Evaluatr.py script (loaded from resource)
      using (var fs =
        new FileStream(filename, FileMode.Create, FileAccess.ReadWrite))
      {
        BinaryWriter bw = new BinaryWriter(fs);
        bw.Write(Resources.Evaluatr);
      }

      evaluatr = env.Runtime.UseFile(filename);

      // Delete the file
      File.Delete(filename);
    }
```

The main **evaluate()** method has the following signature:

```
public IList>
evaluate(string expr,
         IEnumerable values)
```

It accepts a mathematical expression as a string and a list of doubles (the **X** values). It returns a list of pairs of doubles (the **X,Y** pairs). The **evaluate()** method starts by declaring an empty result list and calling the **evaluate** method of the embedded IronPython **evaluatr** module:

```
  var result = new List>();

  //Invoke the method
  var r = evaluatr.evaluate(expr, values);
```

The return value **r** is a dynamic object, which actually contains an **IronPython.Runtime.PythonGenerator** object. This is not the type of object we want to expose to the rest of the program (the idea is to hide the implementation). So, we iterate over the generator and push every pair to the result list.

If the expression is invalid, then an IronPython exception is raised (in Python, you raise exceptions instead of throwing them), and we translate that to a C# exception, which we then throw, as shown next:

```
foreach (var s in r)
{
  try
  {
    double x = s[0];
    double y = s[1];
    var pair = new KeyValuePair(x, y);
    result.Add(pair);
  }
  catch (Exception)
  {
    throw new Exception("Invalid expression: ‘" + expr + "’");
  }
}

return result;
```

The other **evaluate()** method is pretty much the same but it accepts integer values. Internally it converts them to doubles and calls the main method:

```
public IList>
       evaluate(string expr, IEnumerable values)
{
  // Convert integers to doubles
  var xValues = values.Select(i => (double)i);

  // Call the evaluate() method that operates on doubles
  return evaluatr.evaluate(expr, xValues);
}
```

Managed Extension Framework (MEF)

MEF was released as part of the .NET framework 4 and Silverlight 4. It provides a standard way for applications to discover and load extensions and to expose application services to extensions. In addition, the application extensions may depend on each other.

The main parts of MEF are the catalog and the composition container. The catalog is responsible for locating composable parts, and the container is responsible for creating parts and their dependencies (which are other parts). Parts have import and export contracts and they interact with other parts using these contracts. MEF provides an attribute-based programming model, which is very easy to use and yet provides good error messages when things go wrong. For further MEF information, we recommend http://mef.codeplex.com/documentation.

Plugin-based Applications

We prefer the term "plugin" instead of "application extension," so that's what we'll use from now on. My definition of a plugin is a piece of code that has an interface and is loaded by some application dynamically at runtime; then, the application interacts with the plugin through the interface. The application may provide certain services to the plugin though its interface.

The application and the plugins are loosely coupled and interact through interfaces. The application discovers and loads the plugin through some standard mechanism that (often) can be reused by other applications. We usually prefer to have a well-known directory that contains plugins. Developers and/or administrators may add or remove plugins from this directory. The benefit of such a programming model is that it is very easy to evolve an application and develop major parts of its functionality in isolated plugins that are easy to test and deploy and don't require modifying the application code itself. It also makes it very easy to create custom apps with different functionality (just pick a custom set of plugins). Finally, application startup time can be dramatically reduced if plugins are loaded only when they are needed.

Graphr is a plugin-based application. It has a **MainWindow** class that's responsible for interacting with the user, the embedded IronPython evaluator, and configuring the active graph properties. It has a GraphManager class that's responsible for loading the graph plugins and managing the active graph. And it has graph plugins that are responsible for displaying the current graph data.

All the components interact through interfaces and promote decoupling. The interfaces are defined in their own assembly (**Graphr.Contract**) and are referenced by Graphr itself and by each plugin. Graphr and the plugins don't reference each other. Their main interface is **IGraph**. There is an additional interface, **IGraphMetadata**, that is used by MEF.

The IGraph interface

The **IGraph** interface has three methods: **PopulateCanvas()**, **DoGraphLayout()**, and **ConfigSpec**.

The **PopulateCanvas()** method is called whenever the data is modified. It accepts a **Canvas** object (where the graph is displayed), the data as a list of **x,y** values, and a **config** object, which is a dictionary of graph specific items like color and stroke thickness. The graph plugin that implements the interface should construct the visual elements that represent the current data and the graph properties. For example, a bar graph will create a bar object for each x,y pair,and set its border color and fill color based on the graph properties.

The **DoGraphLayout()** method is called every time the graph needs to be redrawn. This happens whenever the canvas is resized (when the whole window is resized or the splitter between the left and right pane is moved). It accepts the canvas and the data. You could argue that the plugin could store the canvas and the data when **PopulateCanvas()** is called — then the **DoGraphLayout()** method could be simpler with no arguments. This is true, but since the **MainWindow** that calls **DoGraphLayout()** must have the canvas and data anyway, it is better not to store the same information in each plugin, too.

The **ConfigSpec** property is a dictionary that maps names to type and object. Each graph plugin supports a different set of named configuration items of different types. A dictionary of typed objects is a generic way to represent this set. The **ConfigSpec** contains the initial values of these items (e.g., **StrokeThickness = 2**). The graph properties pane displays these initial values and the user may modify them and change the appearance of the displayed graph. Here is the entire **IGraph** interface:

```
public interface IGraph
{
  void PopulateCanvas(
    Canvas c,
    IList> data,
    IDictionary config);

  void DoGraphLayout(
    Canvas c,
    IList> data);

  // The config spec is a triplet of config items
  // name: string
  // type: Type
  // initial value: object of the item's type
  IDictionary> ConfigSpec { get; }
}
```

The IGraphMetadata Interface

This is a very simple interface that just has a single property called **Name**:

```
public interface IGraphMetadata
{
  string Name { get; }
}
```

The IGraphMetadata Interface

This is a very simple interface that just has a single property called **Name**:

```
public interface IGraphMetadata
{
  string Name { get; }
}
```

This is an interface used by MEF to annotate plugins with metadata that can be queried without instantiating the plugin. You will see later how this metadata is used by Graphr.

GraphManager and MEF

GraphManager is a class that encapsulates the plugins and the interaction with MEF and also keeps track of the current active graph plugin. It begins by loading the graph plugins, which are implemented as MEF parts. The **GraphManager** has the following data member:

```
[ImportMany]
public IEnumerable> Helpers { get; set; }
```

The **Helpers** collection is decorated with MEF's **[ImportMany]** attribute. It is an enumerable collection of **Lazy** pairs of **IGraph** and **IGraphMetadata**. The **Lazy<T, Metadata>** is an MEF extension of the .NET 4 **Lazy<T>** template. It allows accessing the metadata without instantiating the plugin itself. In order for a graph plugin to be discoverable this way, it must comply with the following requirements:

1. Implement **IGraph**
2. Have an **[Export]** attribute with a type of **IGraph**
3. Have an **[ExportMetadata]** attribute with a "**Name**" property that matches the **IGraphMetadata** interface.

The following code is a snippet from the **LineGraph** plugin:

```
[ExportMetadata("Name", "Line")]
[Export(typeof(IGraph))]
class LineGraph : IGraph
{
  ...
```

MEF can discover and load all compliant plugins into the **Helpers** collection. To locate the plugins, we use MEF's **DirectoryCatalog**. Because we want to scan both the current directory and a "plugins" directory, we use an **AggregateCatalog** and add both the current directory and the "plugins" directory (if we can find it). Once the catalog is ready, we create a **CompositionContainer** with the catalog and call its **composeParts()** method. That tells MEF to do its magic: Scan the directories in the catalog, scan each assembly in these directories for types that comply with the graph plugin requirements, and load them (without instantiation) into the **Helpers** collection.

```
    public GraphManager(Canvas c, Grid g)
    {
      this.canvas = c;
      this.propertyGrid = g;
 // Discover and load graph plugins via MEF magic. All the plugins
 // will automagically populate the Helpers collection

      var catalog = new AggregateCatalog();
      catalog.Catalogs.Add(new DirectoryCatalog
         (Directory.GetCurrentDirectory()));

      // Find the plugins directory
      var cd = Directory.GetCurrentDirectory();
      var d = cd;
      var pluginsDir = Path.Combine(d, "plugins");
      var root = Directory.GetDirectoryRoot(d);
      while (d != root)
      {
        d = Path.GetDirectoryName(d);
        pluginsDir = Path.Combine(d, "plugins");
        if (Directory.Exists(pluginsDir) && pluginsDir != cd)
        {
          catalog.Catalogs.Add(new DirectoryCatalog(pluginsDir));
          break;
        }
      }

      var container = new CompositionContainer(catalog);
      container.ComposeParts(this);
    }
```

When the user selects a new graph type in the graph selector dropdown box (or when initially loading the default graph type), the **GraphManager** is responsible for the switch. The **SwitchGraphHelper()** method is called, which selects from the Helpers collection the graph plugin whose **Metadata.Name** property (defined in the **[ExportMetadata]** attribute) matches the name argument. Then it populates the graph properties pane using the current helper's **ConfigSpec** and repopulates the canvas if there is data, as shown in this code:

```
    public void SwitchGraphHelper(string name)
    {
      this.helper =
          Helpers.Single(h => h.Metadata.Name == name).Value;

      PopulateGraphProperties(helper.ConfigSpec);
      if (this.data != null)
      {
        PopulateCanvas(this.data);
      }
    }
```

The **PopulateGraphProperties()** method populates the graph properties grid in the left pane dynamically based on **ConfigSpec**:

```
public void PopulateGraphProperties(IDictionary> configSpec)
    {
      config = new Dictionary();
      foreach (var kv in configSpec)
      {
        config.Add(kv.Key, kv.Value.Item2);
      }

      var g = this.propertyGrid;

      g.ColumnDefinitions.Clear();
      g.RowDefinitions.Clear();
      g.Children.Clear();

      // Define the Columns
      ColumnDefinition colDef1 = new ColumnDefinition();
      ColumnDefinition colDef2 = new ColumnDefinition();
      g.ColumnDefinitions.Add(colDef1);
      g.ColumnDefinitions.Add(colDef2);

      int row = 0;
      foreach (var pair in configSpec)
      {
        var s = pair.Value;
        RowDefinition rd = new RowDefinition();
        rd.Height = new GridLength(25);
        g.RowDefinitions.Add(rd);

        var label = new Label();
        label.Content = pair.Key;
        g.Children.Add(label);
        var editor = _createEditor(pair.Key, s.Item1, s.Item2);
        g.Children.Add(editor);

        Grid.SetRow(label, row);
        Grid.SetColumn(label, 0);

        Grid.SetRow(editor, row);
        Grid.SetColumn(editor, 1);
```

```
      ++row;
    }
  }
```

The **_createEditor()** method creates an editor for the particular item type. The supported types are: **string**, **Color**, **Int32**, and **Double**. In the current implementation, it is always a **TextBox**, but in general it can be any **UIEllement**. Different item types are handled differently. The **_createEditor()** method attaches a type-specific handler for each item type:

```
private UIElement _createEditor(string name, Type t, object v)
{
  var e = new TextBox() { Name = name, Text = v.ToString() };
  if (t.Name == "string")
   e.TextChanged +=
              new TextChangedEventHandler(_onTextChanged);
  else if (t.Name == "Color")
   e.TextChanged +=
              new TextChangedEventHandler(_onColorChanged);
  else if (t.Name == "Int32")
   e.TextChanged += new TextChangedEventHandler(_onIntChanged);
  else if (t.Name == "Double")
 e.TextChanged +=
              new TextChangedEventHandler(_onDoubleChanged);
  else
  {
    throw new Exception("Unknown type");
  }

  return e;
}
```

Details about Graphr code (unrelated to MEF or IronPython, but including descriptive details about the plugins) are available here. Complete code and build files for the project are available at http://is.gd/blplMd.

Conclusion

Graphr is a fun project that demonstrates the power and ease of use of WPF and explores the polyglot programming world by embedding IronPython in C#. It also shows how simple it is to add plugins to an application with MEF. If you like it, you may find it interesting to further extend it.

Related Links

Using IronRuby in .NET Programs

— Gigi and Saar Sayfan are regular contributors to Dr. Dobb's, most recently authoring a two-part exploration of the PolyArea Project; see (http://drdobbs.com/windows/226700093 and http://drdobbs.com/open-source/227700264).