Dr.Dobb's
THE WORLD OF SOFTWARE DEVELOPMENT

Graphr: Additional Information

A walk through of Graphr's architecture

May 06, 2011
URL:http://www.drdobbs.com/windows/graphr-additional-information/229402979

---

*Note: This article is an addendum to a lengthy article on Graphr, posted in the June digital issue of **Dr. Dobb's Journal** . That article should be consulted to understand the topics discussed here. That article also presents the location of the complete code for this application. — Ed.]*

Managing the active graph

The active graph needs to be populated when the data is changed, and it needs to be updated when the canvas is resized or when the graph properties are modified. The **GraphManager** forwards the heavy lifting to the current graph helper.

The **Update()** method is called when the expression or data range have changed. It computes the data using the IronPython evaluator and then populates the canvas:

```
public void Update(string expr = null, int start = 0, int end = 0)
{
  var data = this.data;
  if (expr != null)
  {
    var numbers = Enumerable.Range(start, end - start + 1);
    var xValues = numbers.Select(i => (double)i);
    data = evaluator.evaluate(expr, xValues);
  }

  PopulateCanvas(data);
}
```

The **DoLayout()** method just forwards the call to the helper using the current data and config:

```
public void DoGraphLayout()
{
  if (data != null)
    helper.DoGraphLayout(this.canvas, this.data);
}
```

The **PopulateCanvas()** method stores the new data and forwards the call to the helper:

```
public void PopulateCanvas(IList<KeyValuePair<double, double>> data)
{
  this.data = data;
  this.helper.PopulateCanvas(canvas, data, this.config);
}
```

When a graph property is changed in an editor, the corresponding event handler extracts the value, updates the **config** dictionary, and calls the **Update()** method. Here is **the _onColorChnaged()** event handler. It takes the string out of the editor **TextBox** (the sender), and it tries to convert every two characters to a hex number (base 16). There should be 4 such numbers that correspond to alpha (transparency) value, red, green, and blue components (each should be **0..FF**):

```
private void _onColorChanged(object sender, TextChangedEventArgs eventArgs)
{
  try
  {
    TextBox tb = sender as TextBox;
    var name = tb.Name;

    var a = Convert.ToByte(tb.Text.Substring(1, 2), 16);
    var r = Convert.ToByte(tb.Text.Substring(3, 2), 16);
    var g = Convert.ToByte(tb.Text.Substring(5, 2), 16);
    var b = Convert.ToByte(tb.Text.Substring(7, 2), 16);

    config[name] = Color.FromArgb(a, r, g, b);

    this.Update();
  }
  catch (Exception)
```

```
    {
      // Ignore exceptions here. Exceptions may happen if the user
      // is in the middle of changing the value.
    }
  }
```

Note that it is OK for exceptions to occur here if the user is in the middle of modifying the value. The intermediate invalid value will just be ignored. If the user never manages to set a valid value, the last valid value will continue to be used.

The MainWindow

Finally, we get to the **MainWindow**. This class is responsible for all the general-purpose UI that is not specific to any graph type. In particular, it is responsible for the graph selector dropdown box, the expression and data range text boxes. When any of them change, it notifies the **GraphManager**, which takes it from there, dynamically populating the graph selector and the initial expression.

The graph selector dropdown needs to contain the names of all the graph plugins. Thanks to MEF, this is trivial. The **GraphManager**'s **Helpers** property contains (in the **Metadata.Name**) all the information. In the constructor, the **MainWindow** iterates over the **Helpers** and populates the graph selector. It then selects the first graph and calls the **GraphManager**'s **SwitchGraphHelper**() method. Finally, it sets an initial expression, so you the user can start playing right away.

Interacting with the GraphManager

The **MainWindow** interacts with the **GraphManager** in the following cases:

1. The visualize button is clicked
2. The size or layout of the **MainWindow** changes
3. The user selects a new graph type

When the visualize button is clicked it means the expression and/or the data has changed. This results in a call to the graph manager's **Update**(). You may ask why expression/range changes should require an explicit button click, while changes to the graph properties are reflected automatically. The answer is that it avoids a lot of intermediate results while the user is still working on the expression or the range. It is a design choice that may change in the future.

```
private void visualizeButton_Click(object sender, RoutedEventArgs e)
{
  var expr = ruleTextBox.Text;
  var values = rangeTextBox.Text.Split(',');
  var start = Convert.ToInt32(values[0]);
  var end = Convert.ToInt32(values[1]);

  graphMan.Update(expr, start, end);
}
```

When the size or the layout is modified (by moving the splitter) the **GraphManager**'s **DoLayout**() method is called (and it doesn't require populating the canvas, so it's much faster):

```
private void Graphr_SizeChanged(object sender, SizeChangedEventArgs e)
{
  graphMan.DoGraphLayout();
}


private void Graphr_LayoutUpdated(object sender, EventArgs e)
{
  graphMan.DoGraphLayout();
}
```

When the user selects a new graph type, the **GraphManager**'s **SwitchGraphHelper**() is called (and will result in an **Update**(), too):

```
private void onGraphChanged(object sender, SelectionChangedEventArgs args)
{
  graphMan.SwitchGraphHelper((string)this.graphSelector.SelectedValue);
}
```

Anatomy of a Graphr plugin

You have seen the Graphr side of things. Now, it's time to look at the plugins themselves. The plugins should implement the IGraph plugin (of course) and they should respond properly for data and graph property changes that will be communicated by calls to their **PopulateCanvas**() method. Let's take a detailed look at the BarGraph plugin, which does a good job.

The **BarGraph** class is annotated with **[Export]** and **[ExportMetadata]** attributes. They enable MEF discovery. The class also implements the IGraph interface:

```
namespace Graphr
{
  [ExportMetadata("Name", "Bar")]
  [Export(typeof(IGraph))]
  class BarGraph : IGraph
```

```
  {
    ...
  }
}
```

In its **PopulateCanvas()** method, the plugin starts by clearing all the children from the canvas. It then iterates over the data and creates a bar for each data point:

```
    public void PopulateCanvas(
      Canvas c,
      IList<KeyValuePair<double, double>> data,
      IDictionary<string, object> config)
    {
      // Clear the previous graph
      c.Children.Clear();
      int count = data.Count;

      // Create brushes to be used by all bars
      var barBrush =
     new LinearGradientBrush((Color)config["BarColor"], Colors.White, 90.0);
      var strokeBrush = new SolidColorBrush((Color)config["StrokeColor"]);

      for (int i = 0; i < count; ++i)
      {
        Rectangle r = new Rectangle();

        // Describes the brush's color using RGB values.
        // Each value has a range of 0-255.
        byte x = (byte)(i * 255 / count);

        r.Fill = barBrush;
        r.Stroke = strokeBrush;
        dynamic thickness = config["StrokeThickness"];
        r.StrokeThickness = thickness;

        // Add the rectangle to the Canvas.
        c.Children.Add(r);
      }
    }
```

There are two interesting lines in the **PopulateCanvas()** method:

```
dynamic thickness = config["StrokeThickness"];
r.StrokeThickness = thickness;
```

The **config** dictionary contains values of type '**object**.' The **PopulateCanvas()** method knows that **StrokeThickness** is a number that can be converted to a double, but there is no direct way to convert it. By assigning it to a C# 4.0 dynamic variable, it can be assigned directly to **r.StrokeThickness** and the conversion takes place automatically. These two lines replaced the following ugly piece of code:

```
        //var thickness = config["StrokeThickness"];
        //if (thickness is int)
        //  r.StrokeThickness = (int)thickness;
        //else if (thickness is uint)
        //  r.StrokeThickness = (uint)thickness;
        //else if (thickness is float)
        //  r.StrokeThickness = (float)thickness;
```

The next interesting method is **DoGraphLayout()**. This is where all the rectangles are placed properly on the canvas. The **BarGraph** fits all the bars exactly in the canvas (with some padding). The space between bars is 20% of each bar's width. Note the nice SQL-like LINQ queries to find the minimum and maximum heights:

```
    public void DoGraphLayout(Canvas c,
                    IList<KeyValuePair<double, double>> data)
    {
      if (data == null)
        return;

      // Use LINQ query dot syntax to find the max value)
      var maxHeight = (from n in data
                       select n.Value).Max();

      var minHeight = (from n in data
                       select n.Value).Min();

      // Scale all values to fit inside the canvas
      var scale = (c.ActualHeight - 20) / (maxHeight - minHeight);
      List<double> values =
          new List<double>(from n in data select n.Value * scale);

      // Create a space of about 20% between each rectangle
      var elements = data.Count;
      var padding = 10;
      var ratio = 5; //ratio between rectangle width and spacer
      var spacerWidth =
    Math.Max((c.ActualWidth-padding) /
             (elements-1 + elements*ratio), 0);
      int i = 0;
      foreach (Rectangle r in c.Children)
      {
```

```
        r.Width = spacerWidth * ratio;
        r.Height = values[i] - minHeight * scale;
        r.SetValue(Canvas.LeftProperty, padding / 2 +
i * (r.Width + spacerWidth));
        r.SetValue(Canvas.BottomProperty, 10.0);
        i++;
      }
    }
```

An interesting WPF nugget in this method is how the bars are positioned in the canvas. A rectangle has a **Width** and **Height** property, but it doesn't have a **Left**, Top, or **Bottom** property. Instead, you position it by calling the **SetValue()** method and passing **Canvas.LeftProperty** and **Canvas.BottomProperty**, which are attached properties. This odd design is actually very smart and allows setting various properties on objects that apply to their relation with their container. Check out http://msdn.microsoft.com/en-us/library/ms749011.aspx to learn more on attached properties.

Finally, the **BarGraph** provides the **ConfigSpec** property. This is a pretty simple dictionary that contains the BarGraph-specific graph properties:

```
    public IDictionary<string, Tuple<Type, object>> ConfigSpec
    {
      get
      {
        var d = new Dictionary<string, Tuple<Type, object>>();
 d.Add("BarColor",
              Tuple.Create<Type, object>(typeof(Color),
              Color.FromRgb(255, 0, 255)));
        d.Add("StrokeColor",
              Tuple.Create<Type, object>(typeof(Color),
              Colors.Black));
        d.Add("StrokeThickness",
              Tuple.Create<Type, object>(typeof(int), 2));
        return d;
      }
    }
```

Taking Graphr to the next level

There are a few natural directions to evolve Graphr. These include:

- More General Services: It would be very useful to add axis, grid, labels, and legend as well scrolling and zooming. Each plugin may implement all these capabilities itself, but it would be a lot of work and will have to be repeated for each plugin. Coming up with some centralized infrastructure will keep plugins relatively simple and yet allow them to benefit from it.
- Type-specific graph properties UI: The current graph properties UI is just a text box for each property. It would be much nicer to be able pick color using a cool color picker and select values and ranges using sliders, spin controls, etc.
- Load existing data sets: Currently, Graphr only supports data that can be defined with equations of one variable. Adding the capability to load arbitrary data sets will make it applicable in many real-world domains.
- Multiple graphs: Displaying graph for multiple data sets can add another dimension.

Reminder: The context for this information and the code for this article are made available in the lead article in *Dr. Dobb's Journal* June 2011 issue.