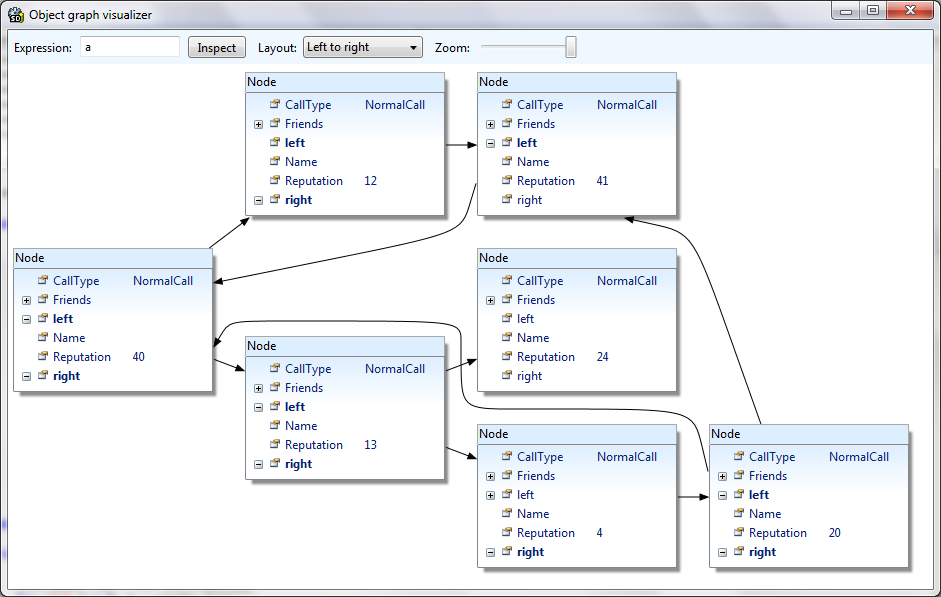
In SharpDevelop 4.1 there is a completely new functionality called Debugger visualizers. This blogpost shows two new visualizers that come as a standard part of SharpDevelop: Collection visualizer and Object graph visualizer.

## Object graph visualizer

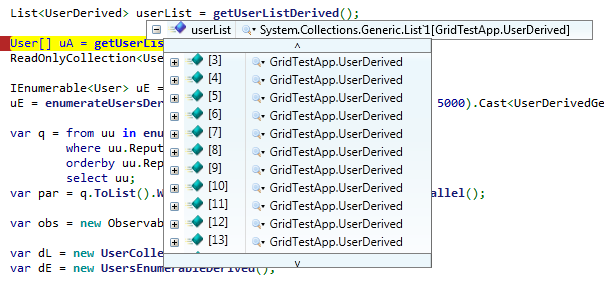
The Object graph visualizer is a bit experimental feature. It draws actual data structures in your program in the same way you would draw data structures say on a whiteboard – boxes connected by arrows. The boxes represent instances and the arrows represent references between them:



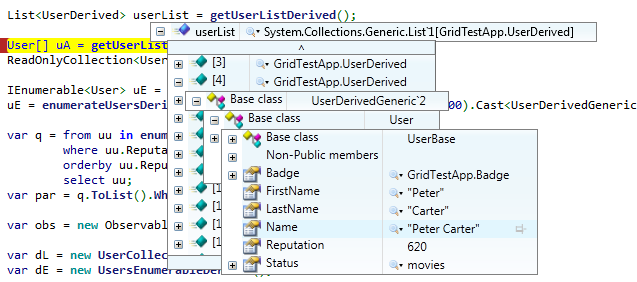
To help you debug algorithms operating on data structures, animated transitions are shown when you step the program in the debugger. This is best explained by a video: embedded youtube smooth video

## Collection visualizer

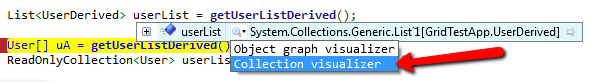
The Collection visualizer solves the following problem: you are debugging a collection of objects in your program and would like to see the contents of the collection. The usual way is the following. You expand the collection variable:

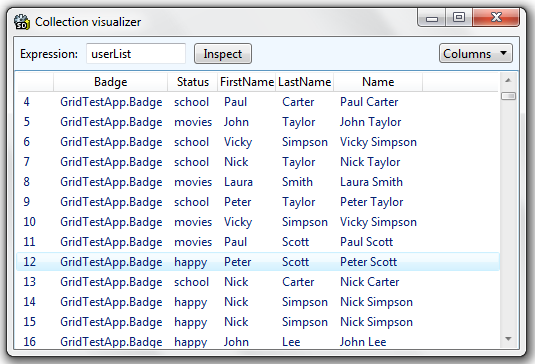


… and see nothing. This view is indeed quite useless. To actually see anything you have to expand individual items:



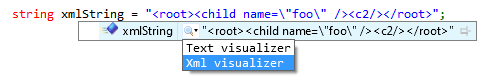
To make matters worse, if the properties you are looking for are declared in base classes you actually have to expand multiple times. Could we do better? I guess we could - enter the Collection visualizer:





This view shows all the properties at once. You can select which properties you want to see using the ComboBox in the top right corner (selecting fewer properties makes scrolling faster).

## Adding your own visualizers

The visualizers are extensible in the sense that you can implement your own visualizers and make them appear in the standard visualizer selector in SharpDevelop. Say we want to implement a visualizer that shows syntax highlighted XML strings:

This is actually quite simple. We need to specify three things:

* When is the visualizer available (e.g. only available for values of type String)
* The label in the UI („Xml visualizer“)
* What should happen when the user executes the visualizer

First we implement an IVisualizerDescriptor which says when the visualizer is available:

**public** class XmlVisualizerDescriptor : IVisualizerDescriptor  
    {  
        **public** **bool** **IsVisualizerAvailable**(DebugType type)  
        {  
            return type.FullName == **typeof**(string).FullName;  
        }  
          
        **public** IVisualizerCommand **CreateVisualizerCommand**(Expression expression)  
        {  
            return **new** **XmlVisualizerCommand**(expression);  
        }  
    }

Now we implement the command which has a label and defines the behavior:

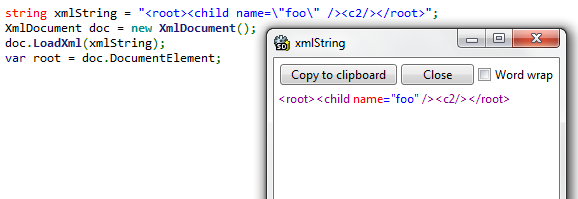
**public** class XmlVisualizerCommand : ExpressionVisualizerCommand  
    {  
        **public** **XmlVisualizerCommand**(Expression expression) :**base**(expression)   { }  
          
        **public** override string **ToString**()  
        {  
            return "Xml visualizer";  
        }  
          
        **public** override void **Execute**()  
        {

// Our visualizer was selected.  
            // We can do anything here - in this case we open a window showing the value of the variable.  
            string expressionValue = **this**.Expression.**Evaluate**(WindowsDebugger.CurrentProcess).**AsString**();  
            var window = **new** **XmlVisualizerWindow**(expressionValue);  
            window.**ShowDialog**();  
        }  
    }

The XmlVisualizerWindow is just a simple WPF Window with a multiline TextBox showing the passed string, highlighted using the built-in syntax highlighting of AvalonEdit. The last thing left is to tell SharpDevelop about our new visualizer. This is done by registering the visualizer in our .addin file:

<Path name="/SharpDevelop/Services/DebuggerService/Visualizers">  
        <Class class="Debugger.AddIn.Visualizers.XmlVisualizerDescriptor" />  
 </Path>

Now SharpDevelop will find the visualizer, display it for String variables and execute our code when the visualizer is selected:



### Note

When executed, a visualizer will typically query some values from the debugger using the debugger API (as we do in Expression.Evaluate) and display the values. The Collection visualizer and Object graph visualizer are implemented exactly in this way as well.

Note that this functionality is not equal to Visual Studio’s debugger visualizers which work by passing serialized objects from the program to the debugger and therefore:

- require types to be Serializable

- cannot perform lazy loading as the whole object has to be serialized and passed at once

Our visualizers work as SharpDevelop AddIns and obtain the values using the debugger API.

## Notes on implementation

Implementing the Collection and Object graph visualizers was quite interesting. To achieve reasonable performance, the Collection visualizer pulls only the values that are actually visible from the debugger (getting values from the debugger is not cheap as it involves interprocess communication for every single value).

In the Object Graph visualizer it was not easy to detect shared references because the garbage collector is moving objects around in memory while you’re trying to determine how the graph actually looks (see David’s post on debugger internals). Another problem was how to layout the graph and draw the edges so that the animations depict datastructure changes in an understandable way.

In case you are interested more, I actually wrote a master thesis about this which covers everything in depth and should be organized in a way that it is easy to find specific topics. You can get it here.

## Conclusion

Hope you will find these features useful. In case of any issues or ideas for improvement (of the visualizers or this article) or if would like to contribute, feel free to contact me (either comment here or drop me an email at martin.konicek gmail).