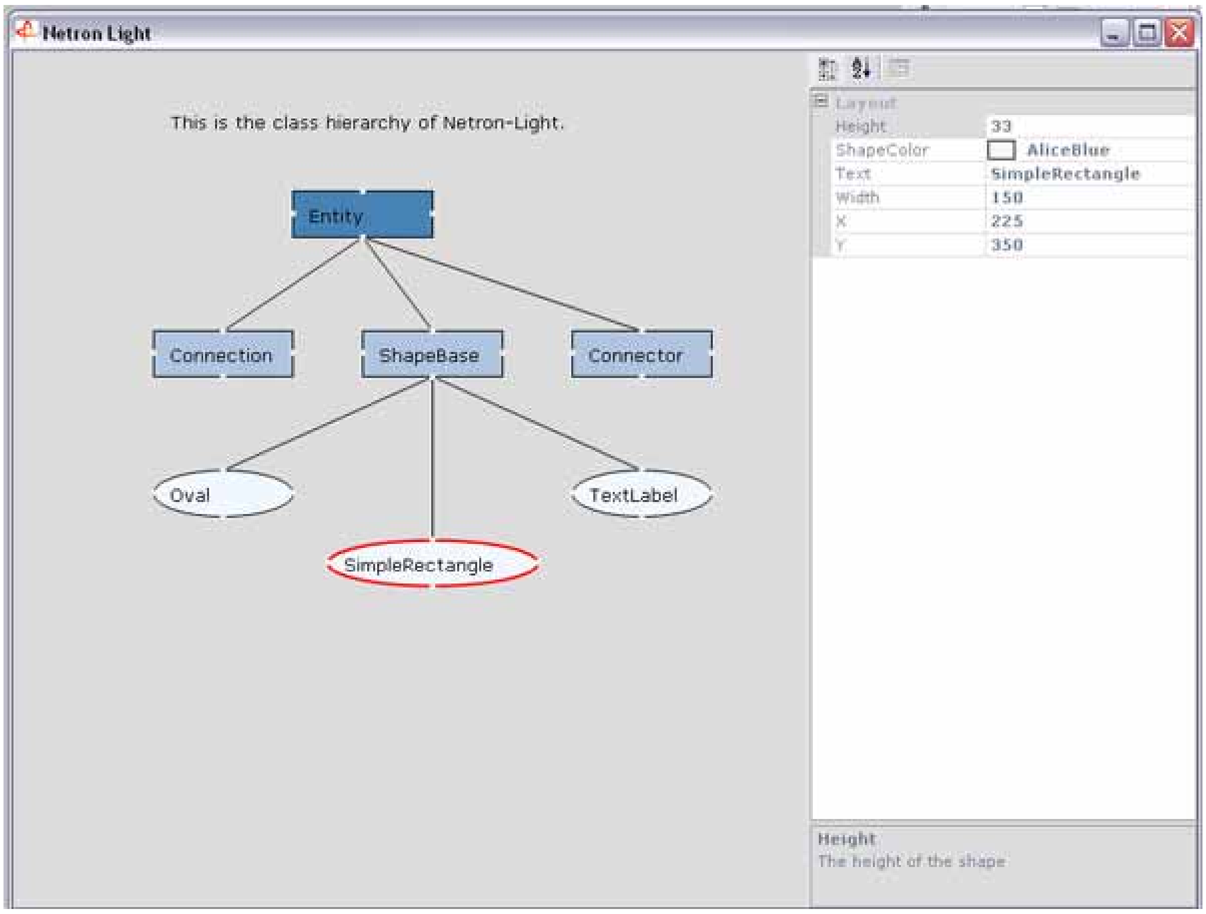
Diagramming for dummies

Abstract:

The structure of a simple diagramming control, called **Netron Light**, is explained in some details. The control is a simplified version of the Netron graph control which has many more features and with an overall more complex architecture. This lightweight version contains however all the core code to make graphs and flow-charts and can be used as a toy-model for a more elaborate control or, if you wish, to help you understand the full version.

Author:
Francois M.Vanderseypen, Illumineo@users.sourceforge.net
The Netron Project, http://netron.sf.net

Monday, 10 January 2005



This is the class hierarchy of Netron-Light.

Layout

Height	33
ShapeColor	⬜ AliceBlue
Text	SimpleRectangle
Width	150
X	225
Y	350

Height
The height of the shape

Introduction

Diagramming data can help you understand this data, look at it from a different perspective and gain insight. Think of Microsoft's BizTalk, Query Analyzer, Data Transformation Services Designer and so on. Diagrams are being used in many ways and for many things, from flow-charting to programming, from network charting to real-time non-linear programming in animations, from GSM technology to GPRS positioning, from simple hierarchies to highly complex academic problems, graphs and diagrams are everywhere. Over the past years topics like mind-mapping ([9]), small-world phenomena (six degrees of freedom [10]) or ontologies (the semantic web [8]) have become popular and are related to diagramming and presenting data in effective, clear ways. On another level graphs are being used to program and track bugs, either to clip away technicalities, to maintain complexity to a certain level or because it simply allows a bigger audience to develop things (cfr. The DTS designer in SQL Server, for example).

Before .Net came along diagramming was solely reserved to the C++ community, it's hard to imagine a simple way to write a diagramming control in Visual Basic 6. In this .Net era it has become (almost) a piece of cake to write a diagramming control. To demonstrate this statement the following article introduces a simple control which allows you to create interactive flow-charts and graphs. More precisely, it allows you to:

- create flow-charts; adding shapes and interactively connect them (though it's allowed to have freely floating connection, i.e. unattached to a shape)
- extend with ease the collection of available shapes and connections
- change properties of the graph canvas and diagram object at run-time by means of the usual property grid (aka 'property browser')
- learn how easy it is to develop GDI+ user-controls

The control does not contain, however, advanced features like:

- printing and saving diagrams (serialization)
- graph layout
- drag-and-drop
- scrolling or constraining diagrams
- grid and grid-snapping
- zoom-unzoom diagrams
- a document object model of the diagram
- selecting multiple diagram objects (the so-called 'marching ants' rectangle to select multiple objects)

Some of these features, like drag-and-drop or printing, are easy to implement and you are encouraged to find out for yourself. Serialization or graph layout algorithms are less evident. This control is, in fact, a lightweight version of the **Netron graph library** ([1]). The Netron library does contain the features listed above and much more. It allows you among other things to:

- program with diagrams (the so-called visual programming paradigm), see the Biotron application ([5]) for an example
- extend the control with shape and connection libraries via a reflection mechanism
- analyze graphs with traditional algorithms, like the Kruskal algorithm, Dijkstra or Depth-first algorithms
- create intelligent shapes with scripting or code generation
- develop applications like Microsoft's Query analyzer, DTS designer and so on

The Netron graph library is fully open source and this article aims at opening up the essential ingredients you'll find in the full graph control. The code accompanying this article can either help you to create your own custom diagramming tool or, if you wish, go deeper into the code of the complete Netron library. In the explanation below I'll point out where this control differs from the complete one. On the other, more features don't necessarily always mean 'better'. If you only need to display, say, some XML in a diagram inside your own application, then why would you carry in a control all the other features? In this sense, this lightweight version of the Netron library fills a gap in the market since versatility and simplicity don't usually co-exist.

I sincerely hope this article will convince you how fun and easy it is to develop your own owner-drawn control and will launch even more interest in using diagrams in WinForm applications.

Comments and suggestions are welcome here at CodeProject or via forum of The Netron Project ([1] and [2]).

A bird's eye view

Before anything else: when referring to the 'canvas' I mean the visible rectangular part of the diagramming control, the thing underneath the diagram. Terms like 'graphs' and 'diagrams', 'node' and 'shape', 'connection' and 'link'… are equivalent, they just emphasize different representations of the same things

Creating a graph is, in essence, really simple; get a reference to a Graphics object and use the available drawing methods to draw. The System.Drawing namespace (and the namespaces underneath) has everything you need to create fancy drawings with ease. So far for the drawing part, the difficulties arise when you want to change the diagram interactively via the mouse:

- You could use the Controls collection of the Panel to add owner-drawn controls and build in this way connected controls, but what control will you use for a connection?
- When the mouse hovers over a shape, how do you get a pointer to it?
- If connections are attached to a shape, how do you communicate to them to move when the shape is moved around?
- When you have moved a shape to a new location, do you need to re-draw it, do you need to erase the previous location?

The idea to use the Label object (Button object would do as well) and connect them via a DrawLine() method of the Graphics class actually works. The problem is that the Label class contains 'too much', it inherits from all its parents; things like drag-drop features, diverse overridable methods which have not really a meaning in a diagramming context.

Concerning the erase and re-draw question; a custom control re-paints itself many, many times a minute. Every time the control executes its OnPaint method the control is cleared and everything is re-painted as if nothing was present before. It means, hence, that if you do not paint it, it will not be visible. If you wish to move a shape around you don't need to erase the previous location, you simply need to update and paint its current position. This picture is a bit rough though since you can invalidate (read: refresh) portions of the control's space but for now this picture is good enough.

The remaining other questions stated above actually come down to use the three basic mouse event handlers in a smart way. We'll review them below and explain the strategy.

When the mouse is pressed the overriden **OnMouseDown** is executed:

- *Either the user will move a diagram-element*; set a flag to be checked in the OnMouseMove handler
- *Either the user selects an object:* find out which object was selected and set its selected bit to true.

How do you know which shape got selected? Well, the shape needs to say it if given a certain coordinate. This is precisely what the **Hit(Point)** methods does. This method, besides a few others is defined in an abstract base class, called the **Entity** class. Herein, all the essential methods are defined in order for a derived class to participate in the diagramming game. When the mouse is pressed various loops go over the diagram elements (shapes, connection , connector) and call the Hit method, the first one saying 'yes' is an (not necessarily unique) object underneath the mouse and all other loop turns are discarded. The variable selectedEntity contains now a pointer to one of the diagram elements. This entity can be of three kinds: shape, connection or connector. It is thanks to the class hierarchy that you can have this; all diagram elements end their inheritance chain in the Entity class.

In the mouse-down event you don't know whether the user wants to select or drag an object. A Boolean field, called '**tracking**', is switched on assuming that a dragging operation is imminent. This field will be set back to false when the user releases the mouse button. If the user moves the mouse in between the switch then the selected diagramming object (set in the selectedEntity) will be moved around. If the user did not want to move something the object has been simply selected and the tracking field is reset to false. In any case, we have fulfilled both situations stated above.

When the user moves the mouse on the canvas the **OnMouseMove** is executed. You can use this method to:

- *Either give feedback to the user* about possible actions
- *Either move a diagram object* selected earlier in the OnMouseDown handler

Feedback to the user is the most important criterion you'll read in any book on software usability. How much feedback is a matter of taste, in our case:

- *Shapes and connections get highlighted when being hovered*
- *Connectors get highlighted when a connection is dragged around*, assuming that the connection will be attached to a shape at its shape-connectors

The way to do this is analog to the looping described above; if, say, a shape return true on a Hit-call, the 'hovered' field of that shape will be set to true, which is subsequently tested in the Painting method of the shape. If true, the paint will additionally paint a red rectangle, otherwise the usual paint occurs.

Finally, when the mouse button is release the **OnMouseUp** handler is executed. As said before, the 'tracking' bit is reset but you need to test whether a connection is being dragged around. If that's the case and the mouse was release above a shape-connector, then the connection has to be added to the collection of connections the shape has. You will notice in the code that actually the connection is not directly connected to a shape but via a shape-connector, see further on this article for more on this.

Here, then, ends the processing loop. Note that in this story the OnMouseMove is called many, many times (just like the OnPaint method of the control). In this perspective it's rather important to have the code inside these methods really compact and efficient although you won't notice a slow-down unless you have huge diagrams or a slow CPU.

Class hierarchy

The class hierarchy reflects the gradual refinement of classes and helps, as is well-known, maintenance and extensibility of a library. The class hierarch of this lightweight version of the Netron graph control is very simple but is a good foundation if you wish to develop more complicated features. In contrast to the full Netron library we don't enforce here any relation between connections and connectors, i.e. connections can freely move on the canvas even if they are not connected to a shape. This approach has advantages but is not viable in the full Netron control since it would make the underlying automata mechanism ill-defined. However, if you wish to create a simple flow-chart model (cfr. Microsoft Visio) these freely moving connections make sense.

We'll review now the four basic classes upon which the graph control relies.

The Entity class

The Entity class is an abstract (MustInherit in VB.Net) base class containing the basic stuff and methods used by all entities drawn on the canvas. It serves as a template, if you wish, for specific objects. Among other things it defines the **Site** property which holds a pointer to the graph control. This pointer is necessary to be able to call methods from the base control class, like e.g. the Invalidate(Rectangle) method.

The abstract methods defined in the Entity class reflect the fundamental methods necessary to participate to the diagramming game:

- **Paint(Graphics):** paints the object on the canvas using an instance of the Graphics object. The Graphics will always be a reference to the canvas of the GraphControl class, i.e. the visible element of the control.
- **Hit(Point):** Each object needs to tell when it's being hit by the mouse. When you press the mouse-button on the canvas and want to select a shape for example, the shape has to give feedback and tell 'yes, I am selected'. Only one object can be selected.
- **Invalidate:** This method is not strictly necessary but helps to improve performance. It says which part of the control needs to be refreshed. You could refresh the whole control every time you modify part of it but this would degrade the responsiveness. Also, if you have a huge diagram it is not necessary to refresh remote parts, only a neighbourhood of the place where things have changed.
- **Move(Point):** When the mouse is dragging a shape you need to tell the class to shift proportionally to the mouse. This method does not use, however, an absolute position. The Point parameter is a shift-vector with respect to the current location. For example, if a shape

is move, the upper-left corner of the underlying shape-rectangle will be shifted by the given point-value.

The connection class

The Connection class contains the code of the connections, i.e. the line between two shapes. Actually, shapes are not connected directly via a line; a connection connects two connectors (see below). The two connectors of a connection are called **From** and **To.** The difference between these connectors is not fundamental, unless you want to differentiate connections with an arrow or you use the connection for transferring data. The painting of the connection is really easy; it simply uses the location of the two connectors to draw a straight line:

```
g.DrawLine(Pens.Black,From.Point,To.Point);
```

The Hit method in this class might seem a bit difficult at first. The straight line is expanded, in a way, to allow to be hit in a neighborhood of itself. Usually it doesn't make sense to have an exact hit, you want to give feedback already when the mouse is near the connection and not really above it and similarly for the mouse position. The expanded line and the neighborhood of the mouse position are rectangles and using the Contains method of the Rectangle class you can use these two rectangles to test whether the mouse rectangle is contained in the line rectangle. Finally, you need an extra if-then-else statement to differentiate the relative position of the line connectors (SouthWest-NorthEast or NorthWest-SouthEast). The mathematics involved is just basic geometry of straight lines in the plane.

```
public override bool Hit(Point p)
{
Point p1,p2, s;
RectangleF r1, r2;
float o,u;
p1 = from.Point; p2 = to.Point;

// p1 must be the leftmost point.
if (p1.X > p2.X) { s = p2; p2 = p1; p1 = s; }

r1 = new RectangleF(p1.X, p1.Y, 0, 0);
r2 = new RectangleF(p2.X, p2.Y, 0, 0);
r1.Inflate(3, 3);
r2.Inflate(3, 3);
//this is like a topological neighborhood
//the connection is shifted left and right
//and the point under consideration has to be in between.

if (RectangleF.Union(r1, r2).Contains(p))
{
if (p1.Y < p2.Y) //SWNE
{
```

```
o = r1.Left + (((r2.Left - r1.Left) * (p.Y - r1.Bottom)) /
(r2.Bottom - r1.Bottom));
u = r1.Right + (((r2.Right - r1.Right) * (p.Y - r1.Top)) / (r2.Top -
r1.Top));
return ((p.X > o) && (p.X < u));
}
else //NWSE
{
o = r1.Left + (((r2.Left - r1.Left) * (p.Y - r1.Top)) / (r2.Top -
r1.Top));
u = r1.Right + (((r2.Right - r1.Right) * (p.Y - r1.Bottom)) /
(r2.Bottom - r1.Bottom));
return ((p.X > o) && (p.X < u));
}
}
return false;
}
```

The connector class

A connector has a double function:

- It represents the end-points of a connection
- It represents a location on a shape where a connection can be attached

How is a connection attached? When you drag one of the end-points of a connections the control will watch you hovering over the various elements of the diagram (see the OnMouseMove handler), if you hit a connector on a shape it will light up notifying you that if you release the mouse the dragged connector will be added to the attached connectors of the highlighted connector. Hence, a connector of a shape has a collection of connectors, called **attachedConnectors.** This field has no meaning if the connector belongs to a connection. You could, in principle, make two different types of connectors to avoid the double usage but we kept the doubling in order to keep the example as simple as possible. Eventually, you could use the same technique to attach connections together at their end-points like attaching a connection to a shape-connector.

The class contains also an **attachedTo** field containing the connector to which it’s being attached. Of course, if equal to null it’s a free-floating connector.

The paint method is just a tiny white rectangle and the hit method uses a slightly bigger rectangle, which becomes red when hit

```
if(hovered)
     g.FillRectangle(Brushes.Red,point.X-5,point.Y-5,10,10);
else
     g.FillRectangle(Brushes.WhiteSmoke,point.X-2,point.Y-2,4,4);
```

Finally, note that the Move method cascades the motion to attached connectors, if it's a shape-connector:

```
for(int k=0; k<attachedConnectors.Count;k++)
      attachedConnectors[k].Move(p);
```

Remember that the given Point parameter is not an absolute position but a relative shift-vector with respect to the current location. The reason for this is simple: when you look at the OnMouseMove handler you'll notice that it calls the Move method but this OnMouseMove handler is called many, many times when you move the mouse. If you'd use an absolute position as an argument the motion would grow exponentially!

The ShapeBase class

The ShapeBase class is a template for diagram shapes, you only need to override a few methods and it'll give you a plugable shape. It defines, on top of the things already present in the Entity class, a few more properties and methods specific to a shape:

- **Connectors:** The connectors collection, i.e. a spot on a shape where connection can be attached
- **Text:** The text displayed on the shape
- **Width and Height:** available via the property grid
- **Resize():** since shape-connectors are attached relative to the width and height of a shape, you need an extra method to move the connectors when the shape width or height is changed.

The **Hit(Point)** method is not independent of the Paint(Graphics) method since the shape should only be highlighted when the mouse is inside the shape's region, which is not necessarily rectangular. Depending on the complexity of your shape this can be sometimes awkward, compare this to the Hit(Point) method of the connection class. It requires sometimes a bit of ingenuity and mathematics to design a simple yet speedy painting and hitting method. Actually, you can check out via diverse analysis and coding-metric tools that the paint method in .Net can be called as much as a hundred-thousand times per minute! A good rule of thumb is to keep the paint method as small as possible, things like color gradients, Bezier curves, text effects…can be easily implemented but come with a price.

The diagramming control

The graph control inherits from the **SrollableControl** class, which is just a scrollable extension of the **Control** class. In fact, in this lightweight version you could use the Control class without any modifications to the code. The ScrollableControl class allows you to have diagrams expanding beyond the edge of the control, in which case the scrollbars will appear. There is a glitch though; if you try it out you'll notice ugly painting effects because the control needs to be invalidated on scrolling. Unfortunately

.Net doesn't have an event to attach to when some scrolling occurs, you need to filter some specific Windows messaging; see the full Netron code for more on this.

Constructor

The constructor sets

- the double buffering: described below
- defines the context menu: allows to interactively add or remove diagram elements
- defines a randomizer: used to generate a random color for newly created shapes and also when adding a new connection
- defines a proxy class for the property grid, see further on

Painting the control

The overridden OnPaint method of the Control class is where you tell how things look like. The control defines two collections, the shapes collection and the connections collection, which are drawn every time the control paints itself. These collection are strongly typed collections to make life easier, you could use a HashTable or an ArrayList however.

The OnPaint method gives you a reference to the Graphics object of the control; if you look at the many methods of the Graphics class you will discover all the basic building blocks of GDI+. Optionally, you can define the quality of the drawing by means of the **SmoothingMode**, set here to the best anti-aliasing quality:

```
Graphics g = e.Graphics;
//use the best quality, with a performance penalty
g.SmoothingMode= System.Drawing.Drawing2D.SmoothingMode.AntiAlias;

//loop over the connections
for(int k=0; k<connections.Count; k++)
{
	connections[k].Paint(g);
	connections[k].From.Paint(g);
	connections[k].To.Paint(g);
}
//similarly, loop over the shapes and draw them
for(int k=0; k<shapes.Count; k++)
{
	shapes[k].Paint(g);
}
```

The first loop paints all the connections whereafter another one paints all the shapes. You can switch these loops, which will result in connections being visible when they are underneath a shape. Note also that we explicitly paint the connectors of the connection, this can be moved to the Paint method of the connection if you wish (a cascading paint).

If you try it out with your own custom controls you will very soon notice that the well-known control flikkering occurs. Usually it's a good idea to switch on double-buffering when designing your own custom controls:

```
SetStyle(ControlStyles.AllPaintingInWmPaint, true);
SetStyle(ControlStyles.DoubleBuffer, true);
SetStyle(ControlStyles.UserPaint, true);
SetStyle(ControlStyles.ResizeRedraw, true);
```

You'll notice that there is also an overriden OnPaintBackground method in the code. This method gives according to the documentation a better performance than drawing a background in the OnPaint method. We have added the possibility to draw a grid in the code (using a static method of the ControlPaint class);

```
if(showGrid)
ControlPaint.DrawGrid(g,this.ClientRectangle,gridSize,this.BackColor
);
```

You could paint a gradient, some text or even a bitmap. If you really make it fancy the responsiveness will be clearly diminished though, here again: if you want it performant, keep it simple.

Actions, menu and properties

Menu

The context menu defined in the graph control allows you to add or remove diagram elements. The code behind this is really plain and simple.

In the full Netron control you can use drag-drop and choose shapes or connection types from a library, much like the 'stencils' in Microsoft Visio. Besides this the plug-in mechanism allows you to expand the control with shape-libraries without re-compiling the graph control.

Actions

The graph control defines a few overloaded methods to add shapes and connections. An example of how to use the various methods can be found in the constructor of the control-hosting form where we have build up the class hierarchy of the control via these methods:

- **AddShape:** this method allows you to add a shape via code from inside the hosting application in contrast to the context-menu mechanism.
- **AddConnection:** this overloaded method allows you to connect shapes or create free-floating connections on the canvas.

```
OvalShape tl = this.graphControl1.AddShape(ShapeTypes.Oval,new 
Point(400,300)) as OvalShape;
      tl.Text = "TextLabel";
      tl.Height = 33;
      tl.ShapeColor = Color.AliceBlue;
```

The important thing to notice here is the fact that you need to pass the graph-control to the diagram objects via the Site property. If you don't an exception will be thrown around the Invalidate() method since all invalidating methods in essence refer to the Invalidate() method of the Control class.

Properties

When you click on any part of the diagram the property grid will show the properties of the selected object. If no diagram object is selected the properties of the canvas will be shown. Showing properties in the standard property grid is really easy; simply pass a class instance to the SelectObject Property of the property grid and you are done. You can tweak the appearance in the grid by means of a few class attributes; the Description, the Category and the Browsable attribute:

```
[Browsable(true), Description("The backcolor of the shape"), 
Category("Layout")]
public Color ShapeColor
{
      get{return shapeColor;}
      set{shapeColor = value; SetBrush(); Invalidate();}
}
```

Much more can said, however, about the grid but I refer you to [4] for more.

In the full Netron library a very useful 'property bag' mechanism ([6]) allows you to have full control over the appearance of object-properties in the property grid. In this lightweight version we haven't used it to keep things simple. However, passing an instance of the graph-control to the property grid gives too much access at run-time. As an intermediate approach a **proxy pattern** was used to clip the control's properties at run-time. The **Proxy** class hides effectively everything you don't want to show at run-time but passes changes to the visible properties down to the control (see any book or text on patterns for more on the Proxy mechanism).

Conclusion

If you made it to this paragraph you will agree that there is ample space for improvements but this article was meant to sharpen your appetite, not to give you a full fledged diagramming control.

Besides the already mentioned features in the introduction you could try out the following (more-or-less easy) things:

- A wider variety of diagram shapes
- Curved or Bezier connections
- Using the System.Reflection namespace, build a class hierarchy of any given assembly by iteratively reflecting the classes and creating a shape for each
- Use a web-service like Google's search-service to create graphs of search results (see e.g. the Groogler application [7])

Hopefully, you found in this article a source of inspiration, maybe you'll use diagrams in your next application?

References

[1] The Netron Graph Library;
http://netron.sourceforge.net/ewiki/netron.php?id=NetronGraphLib

[2] The Netron Project forum;
http://sourceforge.net/forum/?group_id=69788

[3] The Netron Graph Library White Paper and Architecture Paper;
http://netron.sf.net/downloads/Netron_Graph_Library_V2.1_White_Paper.pdf and
http://netron.sf.net/downloads/Netron%20Graph%20Library%20Architecture%20v0.3.pdf

[4] "Make your components really RAD with Visual Studio.Net Property browser", Shawn Burke, MSDN;
http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dndotnet/html/vsnetpropbrow.asp

[5] The Biotron application;
http://netron.sourceforge.net/ewiki/netron.php?id=BiotronHelp

[6] “Bending the .Net property grid at your will”, Tony Allowatt; http://www.codeproject.com/cs/miscctrl/bending_property.asp

[7] The Groogler application; http://netron.sourceforge.net/ewiki/netron.php?id=GrooglerIIHelp

[8] The Wikipedia Encyclopedia; http://en.wikipedia.org/wiki/Ontology_%28computer_science%29

[9] Out of many: Mind Manager from MindJet; http://www.mindjet.com

[10] “Nexus: small worlds and the theory of networks”, Mark Buchanan, see Amazon.com for example