Introduction to GoXam

Northwoods Software

www.nwoods.com

GoXam provides controls for implementing diagrams in your WPF applications. **GoWPF** is the name for the implementation of **GoXam** for WPF on .NET Framework 4.0 or later or on .NET Core 3.0 or later.

For web apps, the successor to GoXam for Silverlight is **GoJS.** Use **GoJS** for creating diagrams in HTML and JavaScript, running entirely in the browser. See more at https://gojs.net.

You can find samples and documentation for **GoXam** in the installation kit, which is merely a ZIP file. The API reference for **GoXam** is at http://www.goxam.com/3.0/apiCore. You can ask questions and search for answers in the forum https://forum.nwoods.com/c/goxam or by e-mail to GoXam at our domain (nwoods.com).

This document assumes a reasonably good working knowledge of WPF and of .NET programming, including generics and Linq for Objects and Linq for XML.

Table of Contents

Summary

The **Diagram** class is a WPF **Control** that fully supports the standard customization features expected in WPF. These features include:

- styling
- templates
- data binding
- use of all WPF elements
- use of WPF layout
- animation
- commands
- printing

Diagrams consist of nodes that may be connected by links and that may be grouped together into groups. All of these parts are gathered together in layers and are arranged by layouts. Most parts are bound to your application data.

Each diagram has a **Model** which interprets your data to determine node-to-node link relationships and group-member relationships.

Each diagram has a **PartManager** that is responsible for creating a **Node** for each data item in the model's **NodeSource** data collection, and for creating or deleting **Link**s as needed.

Each **Node** or **Link** is defined by a **DataTemplate** that defines its appearance and behavior.

The nodes may be positioned manually (interactively or programmatically) or may be arranged by the diagram's **Layout** and by each **Group**'s **Layout**.

Tools handle mouse events. Each diagram has a number of tools that perform interactive tasks such as selecting parts or dragging them or drawing a new link between two nodes. The **ToolManager** determines which tool should be running, depending on the mouse events and current circumstances.

Each diagram also has a **CommandHandler** that implements various commands (such as Delete) and that handles keyboard events.

The **DiagramPanel** provides the ability to scroll the parts of the diagram or to zoom in or out. The **DiagramPanel** also contains all of the **Layer**s, which in turn contain all of the **Part**s (**Node**s and **Link**s).

The **Overview** control allows the user to see the whole model and to control what part of it that the diagram displays. The **Palette** control holds nodes that the user may drag-and-drop to a diagram.

You can select one or more parts in the diagram. The template implementation may change the appearance of the node or link when it is selected. The diagram may also add **Adornment**s to indicate selection and to support tools such as resizing a node or reconnecting a link.

Diagram Models and Data Binding

One of the principal features of XAML-defined presentation is the use of data binding. Practically all controls in the typical application will depend on data binding to get the information to be displayed, to be updated when the data changes, and to modify the data based on user input.

A **Diagram** control, however, must support more complex features than the typical control. The most complex standard controls inherit from **ItemsControl**, which will have a **CollectionView** to filter, sort, and group items into an ordered list. But unlike the data used by an **ItemsControl**, a diagram features relationships between data objects in ways more complex than a simple total ordering of items.

There are binary relationships forming a *graph* of *nodes* and *links.* In similar terminology they may be called *nodes* and *arcs*, or *entities* and *relationships*, or *vertices* and *edges*.

There are grouping relationships, where a *group* contains *members*. They may be used for *part*/*sub-part* containment or for the nesting of *subgraphs*.

We make use of a *model* to discover, maintain, navigate, and modify these relationships based on the data that the diagram is bound to. Each **Diagram** has a model, but models can be shared between diagrams.

To be useful, every model needs to provide ways to do each of the following kinds of activities:

- getting the collection of data
- discovering the relationships in the data in order to build up the model
- updating the model when there are changes to the data
- examining the model and navigating the relationships
- modifying the collection of data, and changing their relationships
- notifying users of the model about changes to the model
- supporting transactions and undo and redo
- supporting data transfer and persistence

Some models are designed to be easier to use or to be more efficient when they have restrictions on the kinds of relationships they support. There are different ways of organizing the data. And

you might or might not have any implementation flexibility in the classes used to implement the data, depending on your application requirements and whether you may modify your application's data schema.

To achieve these goals we provide several model classes in the **Northwoods.GoXam.Model** namespace.

Choosing a Model

There are currently three primary model classes that implement the basic notion of being a diagram model.

TreeModel

The **TreeModel** is the simplest model. It is suitable for applications where the data forms a graph that is a tree structure: each node has at most one *parent* but may have any number of *children*, there are no undirected cycles or loops in the graph, and there is at most one link connecting any two nodes.

If your graph is not necessarily tree-structured, or if you want to support grouping as well as links, you will need to use either **GraphModel** or **GraphLinksModel**.

GraphModel

Use **GraphModel** when each node has a collection of references to the nodes that are connected to that node and are either coming from or going to the node. **GraphModel** permits cycles in the graph, including reflexive links where both ends of the link are the same node. However, there can be at most one link connecting each pair of nodes in a single direction, and there can be no additional information associated with each link.

Grouping in **GraphModel** supports the membership of any node in at most one other node, and cycles in the grouping relationship are not permitted. Hence each subgraph is also a node, and node-subgraph membership forms its own tree-like structure.

GraphLinksModel

If you need to support an arbitrary amount of data for each link, or if you need multiple distinct links connecting the same pair of nodes in the same direction, or if you need to connect links to links, you will need to use a separate data structure to represent each link. The **GraphLinksModel** takes a second data source that is the collection of link data.

GraphLinksModel also supports additional information at both ends of each link, so that one can implement logically different connection points for each node.

GraphLinksModel supports group-membership (i.e. subgraphs) in exactly the same manner that **GraphModel** does.

Model Data

The model classes are generic classes. They are type parameterized by the type of the node data, ***NodeType***, and by the type of the unique key, ***NodeKey***, used as references to nodes. In the case of **GraphLinksModel**, there is also a type parameter for the link data type, ***LinkType***, and a type parameter for optional data describing each end of the link, ***PortKey***. (However, the implementation of diagram **Node**s expects that ***PortKey*** must be a String.)

The model classes can probably be used with your existing application data classes. If you do not already have such data classes you can implement them by inheriting from the optional data classes that are in the **Northwoods.GoXam.Model** namespace, to add application-specific properties.

Generic Models	Suggested data classes
TreeModel <*NodeType*, *NodeKey*>	**TreeModelNodeData<*NodeKey*>**
GraphModel <*NodeType*, *NodeKey*>	**GraphModelNodeData<*NodeKey*>**
GraphLinksModel <*NodeType*, *NodeKey*, *PortKey*, *LinkType*>	**GraphLinksModelNodeData<*NodeKey*>** and **GraphLinksModelLinkData<*NodeKey, PortKey*>** (or **UniversalLinkData**)

The typical usage of models and data is:

```
// create a typed model
var model = new GraphLinksModel<MyData, String, String, MyLinkData>();
// maybe set other model properties too...

// specify the nodes, which includes the subgraph information
model.NodesSource = new ObservableCollection<MyData>() {
    . . .  // supply the node data
  };

// specify the links between the nodes
model.LinksSource = new ObservableCollection<MyLinkData>() {
    . . .  // supply the link data
  };

// have the Diagram use the new model
myDiagram.Model = model;
// after this point all model changes should be in a transaction
```

The node data and link data classes would be defined like:

```
public class MyData : GraphLinksModelNodeData<String> {
  // define node data properties; setters should call RaisePropertyChanged
}
```

```
public class MyLinkData : GraphLinksModelLinkData<String, String> {
  // define link data properties; setters should call RaisePropertyChanged
}
```

GraphModel and **TreeModel** do not have a **LinksSource** property and you would not need to define or use a link data class.

Getting the model data

Each model needs access to the collection of data that it is modeling. This means setting the **IDiagramModel.NodesSource** property. The value must be a collection of node data.

For example, consider the following model initialization:

```
var model = new GraphModel<String, String>();
model.NodesSource = new List<String>() { "Alpha", "Beta", "Gamma", "Delta" };
```

This produces a graph without any links. The node data are just strings. Without any customized templates it might appear as:

In future sections we will discuss customizing the appearance and behavior of nodes using **DataTemplate**s.

If you want to be able to add or remove data from the **NodesSource** collection and have the model (and diagram) automatically updated, you should do the following:

```
model.NodesSource =
  new ObservableCollection<String>() { "Alpha", "Beta", "Gamma", "Delta" };
```

ObservableCollection is in the System.Collections.ObjectModel namespace. It provides notifications when the collection is changed, so **Add**ing a string to that **ObservableCollection** will cause an extra node to be created in the model and shown in the diagram.

Discovering Relationships in the Data

In order to build up the model's knowledge of links between nodes, the model must examine each node data for link information, or it needs to be given link data describing the connections between the node data. Usually that information is stored as property values on the data, so you just need to provide those property names to the model. For generality, not only are simple property names

supported, but also XAML-style property paths, typically property names separated by periods. Thus most model properties that specify property accessor paths have names that end in "**Path**". An example is **NodeKeyPath**, which specifies how to get the key value for a node data object.

However, when the information is not accessible via a property path, perhaps because a method call is required or because the information needs to be computed, you can override protected virtual methods on the model to get the needed information. These discovery-implementation methods have names that start with "**Find**". Because using a property path may use reflection, overriding these methods also produces an implementation that is faster and that is more likely to work in limited-permission environments.

Link information in the node data

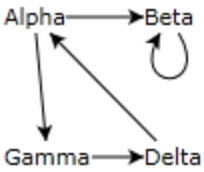
If you have the link relationship information stored on each node data, you might implement the node data class to have a property holding the name of the node and another property or two holding a collection of names that the node is connected to. This is how **GraphModel** expects the information to be organized.

If you don't want to implement your own node data class, you can use one that we provide, **GraphModelNodeData**. This is a generic class, parameterized by the type of the key value. In the following examples, the keys are strings. We just need to specify the property names for discovering the "name" of each node and for discovering the collection of connected node names.

```
// model is a GraphModel using GraphModelNodeData<String> as the node data
// and strings as the node key type
var model = new GraphModel<GraphModelNodeData<String>, String>();

model.NodeKeyPath = "Key";      // use the GraphModelNodeData.Key property
model.ToNodesPath = "ToKeys";   // the node property to get a list of node keys

model.NodesSource = new ObservableCollection<GraphModelNodeData<String>>() {
    new GraphModelNodeData<String>() {
        Key="Alpha",
        ToKeys=new ObservableCollection<String>() { "Beta", "Gamma" }
    },
    new GraphModelNodeData<String>() {
        Key="Beta",
        ToKeys=new ObservableCollection<String>() { "Beta" }
    },
    new GraphModelNodeData<String>() {
        Key="Gamma",
        ToKeys=new ObservableCollection<String>() { "Delta" }
    },
    new GraphModelNodeData<String>() {
        Key="Delta",
        ToKeys=new ObservableCollection<String>() { "Alpha" }
    }
};
```

```
myDiagram.Model = model;
```

The result might appear as:



Link information as separate link data

If you have the link data separate from the node data, as is the case for **GraphLinksModel**, you might do:

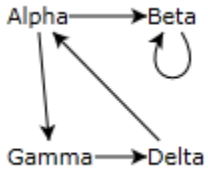
```
// model is a GraphLinksModel using strings as the node data
// and UniversalLinkData as the link data
var model = new GraphLinksModel<String, String, String, UniversalLinkData>();

// the key value for each node data is just the whole data itself, a String
model.NodeKeyPath = "";
model.NodeKeyIsNodeData = true;  // NodeType and NodeKey values are the same!
model.LinkFromPath = "From";  // UniversalLinkData.From → source's node key
model.LinkToPath = "To";  // UniversalLinkData.To → destination's node key

model.NodesSource =
  new ObservableCollection<String>() { "Alpha", "Beta", "Gamma", "Delta" };

model.LinksSource = new ObservableCollection<UniversalLinkData>() {
    new UniversalLinkData("Alpha", "Beta"),
    new UniversalLinkData("Alpha", "Gamma"),
    new UniversalLinkData("Beta", "Beta"),
    new UniversalLinkData("Gamma", "Delta"),
    new UniversalLinkData("Delta", "Alpha")
};

myDiagram.Model = model;
```

Note that the node data in this example are just strings. Because the node value, a string, is also its own key value, there is no property to get the key given a node – hence the **NodeKeyPath** is the empty string. Of course in a "real" application you would have your own node data class, either inheriting from **GraphLinksModelNodeData** or defined from scratch. This would allow you to add all of the properties you need for each node, bindable from the node data templates.

In this example we are using the **UniversalLinkData** class that we provide as a convenient pre-defined class that you can use for representing link data. The **From** property of **UniversalLinkData** is supplied as the first argument of the constructor; it refers to the source node. The **To** property is supplied as the second argument; it refers to the destination node.

UniversalLinkData inherits from **GraphLinksModelLinkData**. As with the node data, the typical "real" application would define its own link data class, either inheriting from **GraphLinksModelLinkData** or defined from scratch, holding whatever information was needed for each link. Defining your own data classes is also more type-safe than using the "Universal…" classes that have properties of type **Object**.

The resulting diagram is exactly the same as for the previous example:



Group information in the node data

Grouping/membership information is accessible in a similar manner, as properties on the node data. For clarity, we use the *subgraph* terminology to refer to groups where each node can have at most one container group. At the current time all GoXam groups are also subgraphs.

You need to set two more model properties used for model discovery:

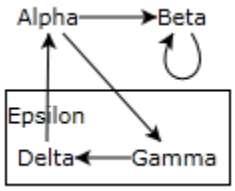
```
// model is a GraphModel or a GraphLinksModel
model.NodeIsGroupPath = "IsSubGraph"; // node property is true if it's a group
model.GroupNodePath = "SubGraphKey";  // node property gets container's name
```

Then change the **NodesSource** data as follows, initializing the two additional properties:

```
// model is a GraphModel using GraphModelNodeData<String> as the node data,
// and the node keys are strings
var model = new GraphModel<GraphModelNodeData<String>, String>();

model.NodeKeyPath = "Key";      // use the GraphModelNodeData.Key property
model.ToNodesPath = "ToKeys";  // this node property gets a list of node keys
model.NodeIsGroupPath = "IsSubGraph"; // node property is true if it's a group
model.GroupNodePath = "SubGraphKey";  // node property gets container's name

model.NodesSource= new ObservableCollection<GraphModelNodeData<String>>(){
  new GraphModelNodeData<String>() {
    Key="Alpha",
    ToKeys=new ObservableCollection<String>() { "Beta", "Gamma" }
  },
  new GraphModelNodeData<String>() {
    Key="Beta",
    ToKeys=new ObservableCollection<String>() { "Beta" }
  },
  new GraphModelNodeData<String>() {
    Key="Gamma",
    ToKeys=new ObservableCollection<String>() { "Delta" },
    SubGraphKey="Epsilon"
```

```
    },
    new GraphModelNodeData<String>() {
      Key="Delta",
      ToKeys=new ObservableCollection<String>() { "Alpha" },
      SubGraphKey="Epsilon"
    },
    new GraphModelNodeData<String>() {
      Key="Epsilon",
      IsSubGraph=true
    },
  };

  myDiagram.Model = model;
```

This results in a diagram that might look like:



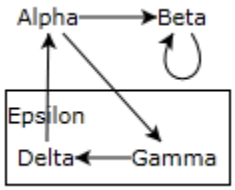
Group information with separate link data

The same result is easily achieved in a **GraphLinksModel** by using **GraphLinksModelNodeData** instead of **GraphModelNodeData** as the node data. In this example we will subclass **GraphLinksModelNodeData** in order to add a property for each node.

```
  // model is a GraphLinksModel using MyData as the node data
  // indexed with strings, and UniversalLinkData as the link data
  var model = new GraphLinksModel<MyData, String, String, UniversalLinkData>();

  model.NodeKeyPath = "Key";  // use the GraphLinksModelNodeData.Key property
  model.LinkFromPath = "From";  // UniversalLinkData.From ➔ source's node key
  model.LinkToPath = "To";  // UniversalLinkData.To ➔ destination's node key
  model.NodeIsGroupPath = "IsSubGraph"; // node property is true if it's a group
  model.GroupNodePath = "SubGraphKey";  // node property gets container's name

  // specify the nodes, which includes subgraph information
  // and other properties specific to MyData, such as Color
  model.NodesSource = new ObservableCollection<MyData>() {
    new MyData() { Key="Alpha", Color="Purple" },
    new MyData() { Key="Beta", Color="Orange" },
    new MyData() { Key="Gamma", Color="Red", SubGraphKey="Epsilon" },
    new MyData() { Key="Delta", Color="Green", SubGraphKey="Epsilon" },
    new MyData() { Key="Epsilon", Color="Blue", IsSubGraph=true },
  };

  // specify the links between the nodes
  model.LinksSource = new ObservableCollection<UniversalLinkData>() {
    new UniversalLinkData("Alpha", "Beta"),
    new UniversalLinkData("Alpha", "Gamma"),
    new UniversalLinkData("Beta", "Beta"),
```

```
    new UniversalLinkData("Gamma", "Delta"),
    new UniversalLinkData("Delta", "Alpha")
  };

  myDiagram.Model = model;


// Define custom node data; the node key is of type String
// Add a property named Color that might change
[Serializable]
public class MyData : GraphLinksModelNodeData<String> {
  public String Color {
    get { return _Color; }
    set {
      if (_Color != value) {
        String old = _Color;
        _Color = value;
        RaisePropertyChanged("Color", old, value);
      }
    }
  }
  private String _Color = "Black";
}
```

This model results in a diagram that looks the same as for the **GraphModel** above. (We'll show how to use the new **Color** property soon.)



If you did not need to support updating the diagram when the value of Color changes, perhaps because you expect the data to be read-only, you could use a simpler implementation of the property:

```
// Define custom node data that does not notify the diagram about Color changes
[Serializable]
public class MyData : GraphLinksModelNodeData<String> {
  public MyData() {
    this.Color = "Black";
  }
  public String Color { get; set; }
}
```

But if you do expect to modify the **MyData.Color** property and expect the corresponding node to change its appearance, you must use the more verbose definition shown earlier that calls **RaisePropertyChanged** in the setter.

Modifying the Model

Once you have created a model, told it how to discover relationships between the nodes (set the various **…Path** properties of the model), initialized the model's data (created a collection of data objects and set the model's **NodesSource**), and assigned the model to your **Diagram**, you might want to programmatically make changes to the diagram. You do this by making changes to the model and to the data, not by trying to change the **Part**s that are in the **Diagram**.

Here's the code for creating a node and a link to that node, given a starting node:

```
// Given a Node, perhaps a selected one, or one that contains a button that
// was clicked, create another Node nearby and connect to it with a new link.
public Node ConnectToNewNode(Node start) {
  MyData fromdata = start.Data as MyData;
  if (fromdata == null) return null;
  // all changes should always occur within a model transaction
  myDiagram.StartTransaction("new connected node");
  // create the new node data
  MyData todata = new MyData();
  // initialize the new node data here...
  todata.Text = "new node";
  todata.Location = new Point(start.Location.X + 250, start.Location.Y);
  // add the new node data to the model's NodesSource collection
  myDiagram.Model.AddNode(todata);
  // add a link to the model connecting the two data objects
  myDiagram.Model.AddLink(fromdata, null, todata, null);
  // finish the transaction
  myDiagram.CommitTransaction("new connected node");
  return myDiagram.PartManager.FindNodeForData(todata, myDiagram.Model);
}
```

Whenever you modify a diagram programmatically, you should wrap the code in a transaction. **StartTransaction** and **CommitTransaction** are methods that you should call either on the **Model** or on the **Diagram**. (The **Diagram**'s methods just call the same named methods on the **DiagramModel**.) Although the primary benefit from using transactions is to group together side-effects for undo/redo, you should use model transactions even if you are not supporting undo/redo.

Note that you do ***not*** create a **Node** directly. Instead you create a data object corresponding to a node, initialize it, and then add it to the model's **NodesSource** collection. It is most convenient to call the model's **AddNode** method, but you could instead insert the data directly into the **NodesSource** collection, assuming the collection implements **INotifyCollectionChanged**.

Programmatically creating links uses the same idea: modify the model by adding or modifying data. For a **GraphModel** or a **TreeModel**, you create a link by setting a node data's property or by adding a reference to a node data's collection of references. For a **GraphLinksModel** you need to create a link data object and insert it into the model's **LinksSource** collection. The **AddLink** method shown

above may work for any kind of model, although for a **GraphLinksModel** there are some restrictions.

How does one get a reference to a node? If you can find the node data object, that's all you need to be able to call **PartManager.FindNodeForData**, as shown above. But if the code is being called from an event handler on an element in a **Node**, you will need to walk up the visual tree until you find the **Node**. The easiest way to do that is with:

```
Node node = Part.FindAncestor<Node>(sender as UIElement);
if (node != null) {
  Node newnode = ConnectToNewNode(node);
  if (newdata != null) {
    // Select the new node
    myDiagram.SelectedParts.Clear();
    newnode.IsSelected = true;
  }
}
```

Data Templates for Nodes

The appearance of any node is determined not only by the data to which it is bound but also the **DataTemplate** used to define the elements of its visual tree.

The simplest useful data templates for nodes are probably:

```
<DataTemplate>
  <TextBlock Text="{Binding Path=Data}" />
</DataTemplate>
```

or:

```
<DataTemplate>
  <TextBlock Text="{Binding Path=Data.Key}" />
</DataTemplate>
```

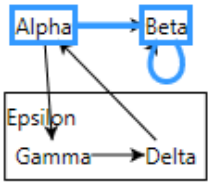
The first one just converts the node's data to a string and displays it; the second one converts the value of the node's data's **Key** property to a string and displays it. The first one is basically the one used in the screenshots shown before that used strings as the node data; the second one was used for those examples that used **GraphModelNodeData** or **GraphLinksModelNodeData** as the node data.

Because templates may be shared, and because it helps to simplify the XAML, you would normally use a node template by defining it as a resource and referring to it as the value of the **NodeTemplate** property of a **Diagram**. For example:

```
<UserControl.Resources>
  <DataTemplate x:Key="NodeTemplate1">
    <TextBlock Text="{Binding Path=Data.Key}" go:Part.SelectionAdorned="True" />
  </DataTemplate>
```

```
  <!-- define other templates here -->
</UserControl.Resources>
. . .
<go:Diagram x:Name="myDiagram" NodeTemplate="{StaticResource NodeTemplate1}" />
```

In this case the node will appear just as with the simpler templates, but when the user clicks on the node, a rectangular selection handle will be appear around the node, visually indicating that it is selected. In the following screenshot, "Alpha" and "Beta" are selected along with the link between them and the link connecting "Beta" with itself.



The node selection effect was achieved just by setting this attached property:

```
go:Part.SelectionAdorned="True"
```

Because **Node** inherits from **Part**, you can refer to precisely the same attached property with:

```
go:Node.SelectionAdorned="True"
```

Setting these kinds of attached properties has to be done on the root visual element of the data template, not on any nested element within that template, nor on the **Node** itself.

In this section we will present more node data templates. These designs concentrate on simple nodes. A later section, "Ports on Nodes", discusses the ability to have links connect to different elements on a node. Other sections discuss data templates for groups and for links.

You can also define multiple templates for nodes and dynamically choose which one to use. This allows you to have many differently appearing nodes in the same diagram. The technique is discussed in a later section about **DataTemplateDictionaries**.

> You can find the XAML for the default templates and styles at:
> `docs\GenericWPF.xaml`

Data Binding

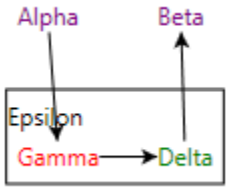
Let's make use of the **MyData.Color** property. In this example each node will have a colored cube as the principal shape, with some text below it. First there needs to be a resource that is a converter from strings (the type of **MyData.Color**) to **Brush:**

```
<go:StringBrushConverter x:Key="theStringBrushConverter" />
```

Then we can bind each text's **Foreground** value to the **Brush** returned by converting the **MyData.Color** property value.

```
<DataTemplate x:Key="NodeTemplate2">
  <TextBlock Text="{Binding Path=Data.Key}"
             Foreground="{Binding Path=Data.Color,
                          Converter={StaticResource theStringBrushConverter}}" />
</DataTemplate>
```

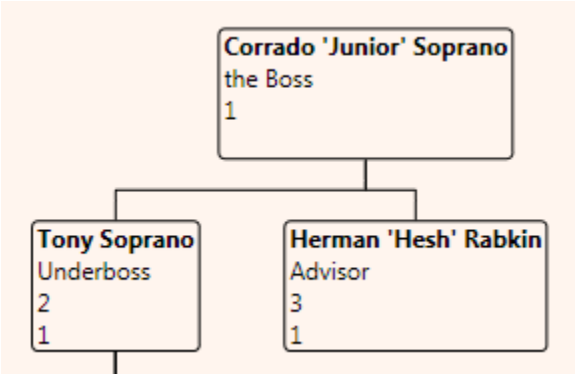
We now get the following screenshot:



Here's a node template consisting of several text blocks surrounded by a border:

```
  <DataTemplate x:Key="NodeTemplate3">
    <Border BorderThickness="1" BorderBrush="Black"
            Padding="2,0,2,0" CornerRadius="3"
            go:Part.SelectionAdorned="True"
            go:Node.Location="{Binding Path=Data.Location, Mode=TwoWay}">
      <StackPanel>
        <TextBlock Text="{Binding Path=Data.Name}" FontWeight="Bold" />
        <TextBlock Text="{Binding Path=Data.Title}" />
        <TextBlock Text="{Binding Path=Data.ID}" />
        <TextBlock Text="{Binding Path=Data.Boss}" />
      </StackPanel>
    </Border>
  </DataTemplate>
```

This results in nodes that look like these three (with an off-white background for the **Diagram**):



Note how the template is bound to the properties of the node data. Most of the bindings are one-way, from the data to the elements. But the binding between the **Node.Location** attached property and the data's **Location** property is two-way: if the value of either property changes, the

other one is updated. This means that not only will modifying the data's **Location** property move the node in the diagram, but interactively dragging the node will modify the data.

For a different kind of node, we will use a **DrawingImage**.

```
<DataTemplate x:Key="NodeTemplateDI">
  <StackPanel go:Part.SelectionAdorned="True"
              go:Node.Location="{Binding Path=Data.Location, Mode=TwoWay}">
    <!-- This image uses a Drawing object for its source -->
    <Image HorizontalAlignment="Center">
      <Image.Source>
        <DrawingImage PresentationOptions:Freeze="True">
          <DrawingImage.Drawing>
            <GeometryDrawing>
              <GeometryDrawing.Geometry>
                <GeometryGroup>
                  <EllipseGeometry Center="50,50" RadiusX="45" RadiusY="20" />
                  <EllipseGeometry Center="50,50" RadiusX="20" RadiusY="45" />
                </GeometryGroup>
              </GeometryDrawing.Geometry>
              <GeometryDrawing.Brush>
                <LinearGradientBrush>
                  <GradientStop Offset="0.0" Color="Blue" />
                  <GradientStop Offset="1.0" Color="#CCCCFF" />
                </LinearGradientBrush>
              </GeometryDrawing.Brush>
              <GeometryDrawing.Pen>
                <Pen Thickness="10" Brush="Black" />
              </GeometryDrawing.Pen>
            </GeometryDrawing>
          </DrawingImage.Drawing>
        </DrawingImage>
      </Image.Source>
    </Image>
    <TextBlock Text="{Binding Path=Data.Text}" HorizontalAlignment="Center" />
  </StackPanel>
</DataTemplate>
```

The example **DrawingImage** was taken from the WPF documentation. This node template just adds a text label centered below the image. The result is:

Using a **DrawingImage** is more resource efficient when the drawing can be shared by multiple nodes.

```
<DataTemplate x:Key="NodeTemplateEG">
  <StackPanel go:Part.SelectionAdorned="True"
              go:Node.Location="{Binding Path=Data.Location, Mode=TwoWay}">
    <Path Stroke="Black" StrokeThickness="10">
      <Path.Fill>
        <LinearGradientBrush>
          <GradientStop Offset="0.0" Color="Blue" />
          <GradientStop Offset="1.0" Color="#CCCCFF" />
        </LinearGradientBrush>
      </Path.Fill>
      <Path.Data>
        <GeometryGroup>
          <EllipseGeometry Center="50,50" RadiusX="45" RadiusY="20" />
          <EllipseGeometry Center="50,50" RadiusX="20" RadiusY="45" />
        </GeometryGroup>
      </Path.Data>
    </Path>
    <TextBlock Text="{Binding Path=Data.Text}" HorizontalAlignment="Center" />
  </StackPanel>
</DataTemplate>
```

Using NodePanel

Of course you can make your templates as complex as you need and as pretty as you want. Because it is common to have each node display some kind of shape along with some text inside it, we have provided the **NodePanel** class which can hold a **NodeShape**. (If you want the text to be outside of the shape, use a **StackPanel** or **Grid** to arrange the elements.)

Furthermore, we have implemented geometries for many common shapes. These are listed by the **NodeFigure** enumeration. By setting the **go:NodePanel.Figure** attached property on the **NodeShape**, the shape will automatically use a **Geometry** corresponding to that particular figure.

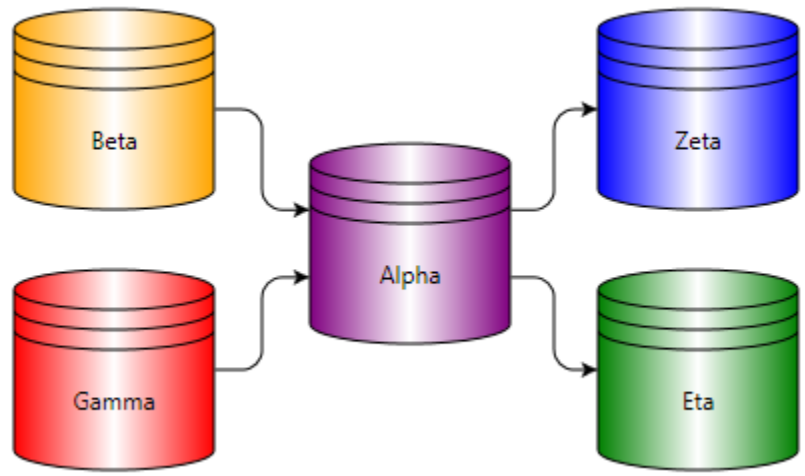
The NodeFigures sample shows all of the predefined shapes, which are enumerated by the **NodeFigure** type.

Consider the following two resource definitions:

```
<!-- define a conversion from String to Color -->
<go:StringColorConverter x:Key="theStringColorConverter" />

<DataTemplate x:Key="NodeTemplate4">
  <!-- a NodePanel shows a background shape and
       places the other panel children inside the shape -->
  <go:NodePanel go:Node.SelectionAdorned="True"
                go:Node.ToSpot="LeftSide" go:Node.FromSpot="RightSide" >
    <!-- this shape gets the geometry defined by the NodePanel.Figure attached
         property -->
    <go:NodeShape go:NodePanel.Figure="Database"
                  Stroke="Black" StrokeThickness="1">
      <Shape.Fill>
        <!-- use a fancier brush than a simple solid color -->
        <LinearGradientBrush StartPoint="0.0 0.0" EndPoint="1.0 0.0">
          <LinearGradientBrush.GradientStops>
```

```
            <GradientStop Color="{Binding Path=Data.Color,
                    Converter={StaticResource theStringColorConverter}}"
                Offset="0.0" />
            <GradientStop Color="White" Offset="0.5" />
            <GradientStop Color="{Binding Path=Data.Color,
                    Converter={StaticResource theStringColorConverter}}"
                Offset="1.0" />
          </LinearGradientBrush.GradientStops>
        </LinearGradientBrush>
      </Shape.Fill>
    </go:NodeShape>
    <!-- this TextBlock element is arranged inside the NodePanel's shape -->
    <TextBlock Text="{Binding Path=Data.Key}" TextAlignment="Center"
               HorizontalAlignment="Center" VerticalAlignment="Center" />
  </go:NodePanel>
</DataTemplate>
```

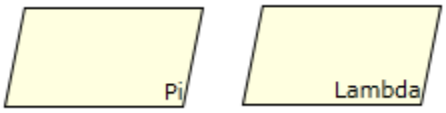
Note how the **LinearGradientBrush** is constructed, binding two of the gradient stop colors to the **MyData.Color** property. The binding also depends on the presence of a **StringColorConverter** (not a **StringBrushConverter**), which was also defined as a resource. The result might look like:



The above use of **NodePanel** assumes that the shape, the first child of the panel, has a fixed width and height. (If the **Width** or **Height** are not supplied, they default to 100, as you can see in the "database" shapes above.) The other children of the **NodePanel** are arranged inside the first child, observing the **HorizontalAlignment** and/or **VerticalAlignment** properties of the child if the width and/or height are smaller than the available area inside the first child. For example:

```
<DataTemplate x:Key="NodeTemplate2">
  <go:NodePanel Sizing="Fixed">
    <go:NodeShape go:NodePanel.Figure="Parallelogram1"
          Width="100" Height="50"
          Stroke="Black" StrokeThickness="1" Fill="LightYellow" />
    <TextBlock Text="{Binding Path=Data.Key}" TextAlignment="Left"
               HorizontalAlignment="Right" VerticalAlignment="Bottom" />
  </go:NodePanel>
</DataTemplate>
```

This might produce:

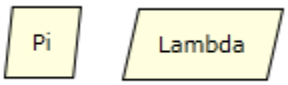


NodePanel.Sizing defaults to **Fixed**. Note the setting of **Width** and **Height** of the shape.

But you can also have the first child be sized to fit around the other children. This is convenient when you want to show a variable amount of text and want the minimal amount of shape surrounding it. Just set **Sizing** to **Auto**:

```
<DataTemplate x:Key="NodeTemplate3">
  <go:NodePanel Sizing="Auto">
    <go:NodeShape go:NodePanel.Figure="Parallelogram1"
          Stroke="Black" StrokeThickness="1" Fill="LightYellow" />
    <TextBlock Text="{Binding Path=Data.Key}" TextAlignment="Left"
               Margin="10" />
  </go:NodePanel>
</DataTemplate>
```

This might produce:



Note how we do *not* set the **Width** or **Height** of the shape. Furthermore we do not set the **HorizontalAlignment** or **VerticalAlignment**, because those properties have no effect. (**TextAlignment** affects how the text is rendered in its allotted space, not how it is positioned in its panel. **Margin** reserves some room around the **TextBlock** – without it the parallelogram would be tightly around the text.)

Resizing

If you want to let users resize such nodes, you first need to think about which element is the "main" element that will control the size and layout of all of the other elements. The "main" element may very well not be the outermost or "root" visual element of the template. So it is not always sufficient to just set `go: Part.Resizable="True"` on the root element; you also need to indicate which element should be the one to get the resize handles and be resized by the **ResizingTool**:

```
<DataTemplate x:Key="NodeTemplate4">
  <go:NodePanel Sizing="Fixed"
                go:Part.Resizable="True" go:Part.ResizeElementName="Shape">
    <go:NodeShape x:Name="Shape" go:NodePanel.Figure="Parallelogram1"
                  Width="100" Height="50"
                  Stroke="Black" StrokeThickness="1" Fill="LightYellow" />
```

```
    <TextBlock Text="{Binding Path=Data.Key}" Margin="10"
               HorizontalAlignment="Center" VerticalAlignment="Center" />
  </go:NodePanel>
</DataTemplate>
```

Note how the root element refers to the **NodeShape** by name, so that user resizing will actually change the width and height of that shape. If you do not specify the **Part.ResizeElementName**, the root element will get the resize handles and attempts to resize the **NodePanel** are not likely to have the effect you want.

Sizing="Fixed" is appropriate because that causes the **NodePanel** to fit the other child elements within the shape. **Sizing="Auto"** causes **NodePanel** to fit the shape around all of the other children, which is not what the user would want if they were trying to resize it. In this case, if you want the user to interactively resize the node, you will need to have the **Part.SelectionElementName** refer to a different child of the **NodePanel**, not the first child.

Collapsing and Expanding Trees

A common technique for simplifying tree-structured graphs is to collapse subtrees. One way to implement this functionality is to add a **Button** to each node.

```
<!-- show either a "+" or a "-" as the Button content -->
<go:BooleanStringConverter x:Key="theButtonConverter"
                           TrueString="-" FalseString="+" />

<DataTemplate x:Key="NodeTemplate">
  <StackPanel Orientation="Horizontal" go:Part.SelectionAdorned="True"
              go:Node.IsTreeExpanded="False">
    <!-- go:Node.IsTreeExpanded="False" tells the node to start collapsed -->
    <go:NodePanel Sizing="Auto">
      <go:NodeShape go:NodePanel.Figure="Ellipse"
          Fill="{Binding Path=Data.Color,
                         Converter={StaticResource theBrushConverter}}" />
      <TextBlock Text="{Binding Path=Data.Color}" />
    </go:NodePanel>
    <Button x:Name="myCollapseExpandButton" Click="CollapseExpandButton_Click"
            Content="{Binding Path=Node.IsExpandedTree,
                              Converter={StaticResource theButtonConverter}}"
            Width="20" />
  </StackPanel>
</DataTemplate>
```

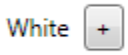
Note that the **Button.Content** is bound to the **Node.IsExpandedTree** property, via a converter that converts the boolean value to either the string "+" or the string "-". Of course you can (and probably should) style the **Button** the way you want instead of using those two text strings. But we'll keep it simple in this document.

The **Button.Click** event handler might be implemented as:

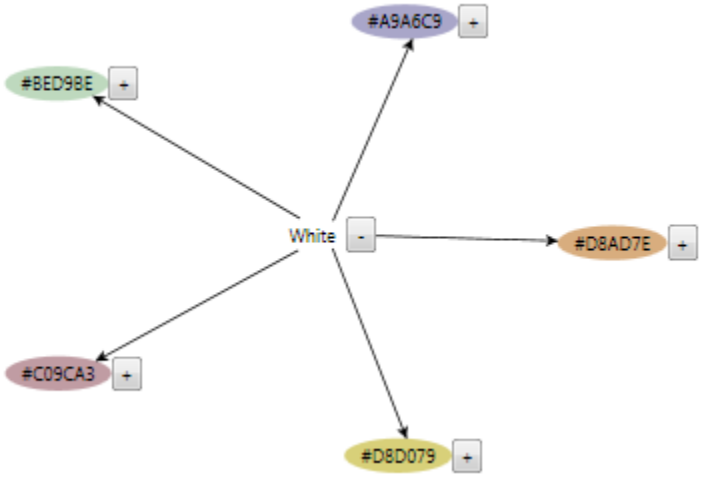
```
private void CollapseExpandButton_Click(object sender, RoutedEventArgs e) {
  // the Button is in the visual tree of a Node
```

```
  Button button = (Button)sender;
  Node n = Part.FindAncestor<Node>(button);
  if (n != null) {
    SimpleData parentdata = (SimpleData)n.Data;
    // always make changes within a transaction
    myDiagram.StartTransaction("CollapseExpand");
    // toggle whether this node is expanded or collapsed
    n.IsExpandedTree = !n.IsExpandedTree;
    myDiagram.CommitTransaction("CollapseExpand");
  }
}
```

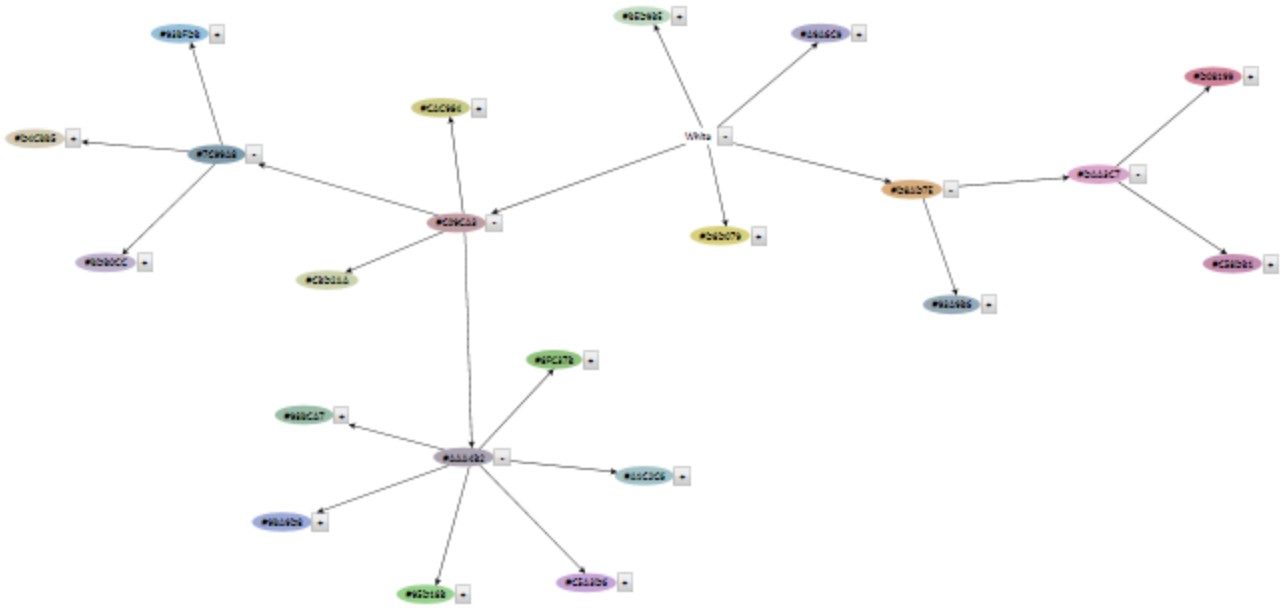
A graph might start with a single node:



An expansion (and a control-mouse-wheel zoom-out) might produce:



Further expansions (and zoom outs) might produce:



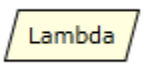
In-place Text Editing and Validation

When you want to let users modify the text in a node, one possibility is to implement your node's **DataTemplate** to have its own **TextBox** that is normally **Collapsed** but that you make **Visible** when you want to edit. In fact, you can have arbitrarily complex controls in each of your nodes. However, the disadvantage is that all of those controls will always be created for each node, thereby increasing the overhead.

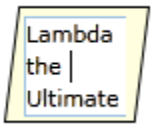
GoXam supports in-place text editing. Just set the **Part.TextEditable** attached property on a **TextBlock**.

```
<DataTemplate x:Key="NodeTemplate5">
  <go:NodePanel Sizing="Auto">
    <go:NodeShape go:NodePanel.Figure="Parallelogram1"
          Stroke="Black" StrokeThickness="1" Fill="LightYellow" />
    <TextBlock Text="{Binding Path=Data.Text, Mode=TwoWay}"
               TextWrapping="Wrap" TextAlignment="Left" Margin="5"
               go:Part.TextEditable="True" />
  </go:NodePanel>
</DataTemplate>
```

(Note that since we expect the user to modify the text, we data bind the text to a different property on the data, not the unique Key.) If the user starts with:



Then if they select the node and then click on the text, the **TextEditingTool** brings up a **TextBox**. The user can edit the text. Losing focus by clicking elsewhere or by tabbing will accept the changes; typing ESCAPE will cancel the edit and restore the original string.



Here you can see the (blinking) cursor positioned at the end of the second line.

You can implement custom text validation by customizing the **TextEditingTool**. This example checks whether the user has typed the letter 'e':

```
public class CustomTextEditingTool : TextEditingTool {
  protected override bool IsValidText(string oldstring, string newstring) {
    bool valid = !newstring.Contains("e");
    if (!valid) {
      MessageBox.Show("Oops: new string contains 'e'");
    }
    return valid;
  }
}
```

and install with either:

```
myDiagram.TextEditingTool = new CustomTextEditingTool();
```

or:

```
<go:Diagram . . .>
  <go:Diagram.TextEditingTool>
    <local:CustomTextEditingTool />
  </go:Diagram.TextEditingTool>
</go:Diagram>
```

From that predicate you can use the **AdornedPart.Data** property to access the bound data.

Spots

Although previous examples have used standard named values such as **Spot.BottomRight** and **Spot.MiddleLeft**, spots are more general than that. A spot represents a relative point from (0,0) to (1,1) within a rectangle from the top-left corner to the bottom-right corner, plus an absolute offset.

Here's a demonstration showing nine text objects positioned at the standard nine spots. This makes use of the **SpotPanel** panel. You may find the **SpotPanel** useful when you want to position smaller elements "inside" another element.

```
<go:SpotPanel>
```

```
  <Rectangle go:SpotPanel.Main="True" Fill="LightCoral"
             Width="200" Height="100" />
  <TextBlock go:SpotPanel.Spot="0.0 0.0" Text="0 0" />
  <TextBlock go:SpotPanel.Spot="0.5 0.0" Text="0.5 0" />
  <TextBlock go:SpotPanel.Spot="1.0 0.0" Text="1 0" />
  <TextBlock go:SpotPanel.Spot="0.0 0.5" Text="0 0.5" />
  <TextBlock go:SpotPanel.Spot="0.5 0.5" Text="0.5 0.5" />
  <TextBlock go:SpotPanel.Spot="1.0 0.5" Text="1 0.5" />
  <TextBlock go:SpotPanel.Spot="0.0 1.0" Text="0 1" />
  <TextBlock go:SpotPanel.Spot="0.5 1.0" Text="0.5 1" />
  <TextBlock go:SpotPanel.Spot="1.0 1.0" Text="1 1" />
</go:SpotPanel>
```

The **SpotPanel.Spot** attached property specifies where the element should be positioned in a **SpotPanel**. The **SpotPanel.Alignment** attached property specifies what point of the element should be positioned at the **SpotPanel.Spot** point. By default the center of each element is aligned at the spot point.

The **Main** attached property says that the spots are all relative to the bounds of the first child element of the **SpotPanel**, which in this case is a **Rectangle**.

Instead of always centering the element at the spot point, you can use any other spot in that element. The following three child elements are all positioned at the same (0, 0) spot, but with different alignments.

```
<go:SpotPanel>
  <Rectangle go:SpotPanel.Main="True" Fill="LightCoral"
             Width="200" Height="100" />
  <TextBlock go:SpotPanel.Spot="0 0"
             go:SpotPanel.Alignment="1.0 1.0" Text="1 1" />
  <TextBlock go:SpotPanel.Spot="0 0"
             go:SpotPanel.Alignment="0.5 0.5" Text="0.5 0.5" />
  <TextBlock go:SpotPanel.Spot="0 0"
             go:SpotPanel.Alignment="0.0 0.0" Text="0 0" />
</go:SpotPanel>
```

Finally, **Spot**s can have absolute offsets in addition to the fractional relative position. These offsets may be negative. You can specify the X and Y offsets as the third and fourth numbers. In this example there are three **TextBlock**s at the bottom-left corner. All have the default center alignment. One has an X offset of negative 30 (i.e. further towards the left), one is centered exactly at the bottom-left corner of the rectangle, and one is shifted towards the right by 30. Similarly there are three **TextBlock**s at the bottom-right corner, with one shifted up 10, and with one shifted down 10.

```
<go:SpotPanel>
  <Rectangle go:SpotPanel.Main="True" Fill="LightCoral"
             Width="200" Height="100" />
  <TextBlock go:SpotPanel.Spot="0 1 -30 0" Text="-30 0" />
  <TextBlock go:SpotPanel.Spot="0 1 0 0"   Text="0 0" />
  <TextBlock go:SpotPanel.Spot="0 1 30 0"  Text="30 0" />
  <TextBlock go:SpotPanel.Spot="1 1 0 -10" Text="0 -10" />
  <TextBlock go:SpotPanel.Spot="1 1 0 0"   Text="0 0" />
  <TextBlock go:SpotPanel.Spot="1 1 0 10"  Text="0 10" />
</go:SpotPanel>
```

Data Templates for Links

The simplest kind of link consists of only a line, perhaps consisting of multiple segments and curves. You must use the **LinkShape** element for this:

```
  <DataTemplate>
    <go:LinkShape Stroke="Black" StrokeThickness="1" />
  </DataTemplate>
```

Like node templates, the typical pattern is to define templates as resources, and refer to them when initializing the **Diagram**:

```
<UserControl.Resources>
  <DataTemplate x:Key="LinkTemplate1">
```

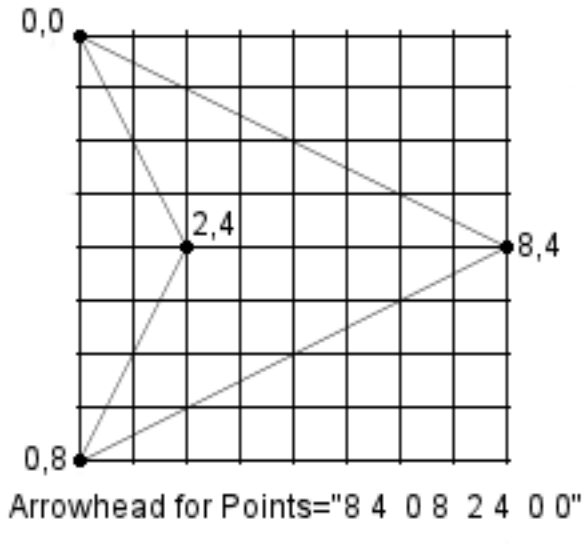
```
    <go:LinkShape Stroke="Black" StrokeThickness="1" />
  </DataTemplate>
  <!-- define other templates here -->
</UserControl.Resources>
. . .
<go:Diagram x:Name="myDiagram" LinkTemplate="{StaticResource LinkTemplate1}" />
```

But note that such a link template will result in links for which there is no arrowhead nor any other decoration. Thus such a simple template can only be used where the links are not directional or where the direction is implicit in the diagram, such as in a tree.

It is more common to have at least an arrowhead on each link. For example, the following template is similar to the default link template – the one used when you do not specify the **Diagram.LinkTemplate** property.

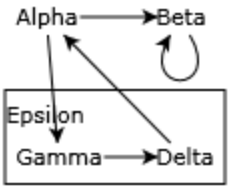
```
<DataTemplate x:Key="LinkTemplate2">
  <go:LinkPanel go:Part.SelectionElementName="Path"
                go:Part.SelectionAdorned="True" >
    <go:LinkShape x:Name="Path" go:LinkPanel.IsLinkShape="True"
                  Stroke="Black" StrokeThickness="1" />
    <Polygon Fill="Black" Points="8 4  0 8  2 4  0 0"  <!-- the arrowhead -->
             go:LinkPanel.Alignment="MiddleRight" go:LinkPanel.Index="-1"
             go:LinkPanel.Orientation="Along" />
  </go:LinkPanel>
</DataTemplate>
```

Here is a visual representation of the points of the polygon:



Arrowhead for Points="8 4 0 8 2 4 0 0"

This results in links that appear like those with arrowheads shown before:



Note the use of the **LinkPanel** class. A **LinkPanel** is a **Panel** that should have a **LinkShape** in it named "Path". The path's **Geometry** is computed by the link's **Route** – i.e. it is given a set of points so that the link shape appears to connect the link's two nodes.

Once the link's route is determined, the **LinkPanel** can arrange all of the other child elements of the panel to be somewhere along the path of the link. In this template, there is a **Polygon** that is acting as an arrowhead. There are three attached properties that control how a **LinkPanel** child such as this **Polygon** is positioned and rotated relative to the link shape. All three are used in this example.

- The **LinkPanel.Alignment** attached property is a **Spot** that indicates what point within the polygon should be positioned along the link path. (More about spots later.) In the above case the MiddleRight spot happens to be the point 8,4.
- The **LinkPanel.Index** attached property specifies at which segment the child element should be placed; zero means at the end near the "from" node, -1 means at the end near the "to" node.
- The **LinkPanel.Orientation** attached property controls whether and how the child element is rotated; "Along" means at the same angle as that link segment.

As a practical matter most link templates consist of a **LinkPanel** holding a **LinkShape** and some varying number of decorations positioned along the link path.

Setting the **Part.SelectionElementName** attached property indicates which element should get a selection handle when the part becomes selected. In this case the link shape will get the selection handle, which is what you would normally want. If you did not set the **SelectionElementName**, the user would see a big **Rectangle** surrounding the whole link, which is probably not what you want.

You can have as many arrowheads as you like. For example, here's a double-headed link:

```
<DataTemplate x:Key="LinkTemplate9">
  <go:LinkPanel go:Part.SelectionElementName="Path">
    <go:LinkShape x:Name="Path" go:LinkPanel.IsLinkShape="True"
                  Stroke="Black" StrokeThickness="1" />
    <!-- the "to" arrowhead -->
    <Polygon Fill="Black" Points="8 4  0 8  2 4  0 0"
             go:LinkPanel.Alignment="1 0.5" go:LinkPanel.Index="-1"
             go:LinkPanel.Orientation="Along" />
    <!-- the "from" arrowhead -->
    <Polyline Stroke="Black" StrokeThickness="1"
```

```
              Points="7 0  0 3.5  7 7"
              go:LinkPanel.Alignment="0 0.5" go:LinkPanel.Index="0"
              go:LinkPanel.Orientation="Along" />
  </go:LinkPanel>
</DataTemplate>
```

This might look like:

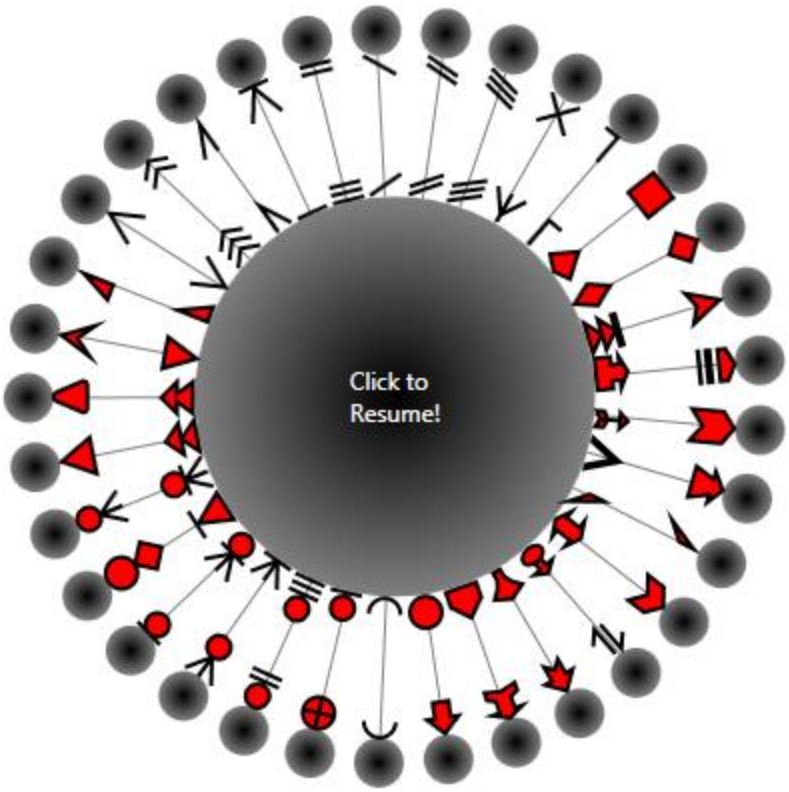
With this mechanism you can implement any arrowhead that you like. The arrowhead element need not be a **Polygon** but can be as complicated as you want. However, this general mechanism isn't so convenient to use.

Therefore we have predefined a number of common arrowheads. You have to provide a **Path** element as an immediate child of the **LinkPanel**, and naturally you can specify its **Fill** and **Stroke** properties. Then you can just set the attached property **LinkPanel.ToArrow**. For example, the following XAML is the same as the above "to" arrowhead element:

```
    <Path Fill="Black" go:LinkPanel.ToArrow="Standard" />
```

You can also change the size of the arrowhead by setting the **LinkPanel.ToArrowScale** attached property. And you can also set **FromArrow** and **FromArrowScale**.

All of the arrowheads are shown by the Arrowheads sample. Note that this screenshot may be out-of-date; look at the **Arrowhead** enumerated type for the complete list.



Of course link templates can be complicated too. If you are using a **GraphLinksModel**, you can bind to the link data. Let's add a text element with a white background:

```
<DataTemplate x:Key="LinkTemplate3">
  <go:LinkPanel go:Part.SelectionElementName="Path"
                go:Part.SelectionAdorned="True">
    <go:LinkShape x:Name="Path" go:LinkPanel.IsLinkShape="True"
                  Stroke="Black" StrokeThickness="1" />
    <!-- the arrowhead -->
    <Polygon Fill="Black" Points="8 4  0 8  2 4  0 0"
             go:LinkPanel.Alignment="1 0.5" go:LinkPanel.Index="-1"
             go:LinkPanel.Orientation="Along" />
    <!-- when using a GraphLinksModel, bind to MyLinkData.Cost as a label -->
    <StackPanel Background="White">
      <TextBlock Text="{Binding Path=Data.Cost}" Foreground="Blue" />
    </StackPanel>
  </go:LinkPanel>
</DataTemplate>
```

This makes use of a **MyLinkData** type that you might define with a **Cost** property:

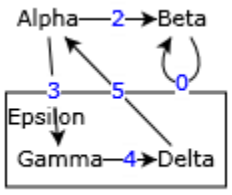
```
[Serializable]
public class MyLinkData : GraphLinksModelLinkData<String, String> {
  public double Cost {
    get { return _Cost; }
    set {
      if (_Cost != value) {
```

```
        double old = _Cost;
        _Cost = value;
        RaisePropertyChanged("Cost", old, value);
      }
    }
  }
  private double _Cost;
}
```

If you expect the link data not to change and need to update the diagram, you could have a simpler implementation of the link data class:

```
[Serializable]
public class MyLinkData : GraphLinksModelLinkData<String, String> {
  public double Cost { get; set; }  // if setter does not need to notify
}
```

The result might look like:



Data Binding to Link Nodes

The example above performed data binding of a **TextBlock**'s **Text** property to a property on the **Link**'s **Data** (an instance of **MyLinkData**). However, it is also possible to data bind link properties to properties on either of the **Link**'s connected **Node**s. This will work even if the model does not support separate link data.

For instance, if you want each link to be colored according to some property of the "To" node, you can bind the Stroke to `{Binding Path=Link.ToData.SomeProperty, Converter={StaticResource someConverter}}`.

For example, we can customize the link colors of the DoubleTree sample by changing the link's template to depend on the **Info.LayoutId** property, where **Info** is a node data class defined in that sample, and where the **LayoutId** property indicates which direction the tree is growing at that node.

```
<local:LinkBrushConverter x:Key="theLinkBrushConverter" />
<DataTemplate x:Key="LinkTemplate">
  <go:LinkPanel>
    <go:LinkShape StrokeThickness="1"
          Stroke="{Binding Path=Link.ToData.LayoutId,
                           Converter={StaticResource theLinkBrushConverter}}" />
    <Polygon Fill="{Binding Path=Link.ToData.LayoutId,
```

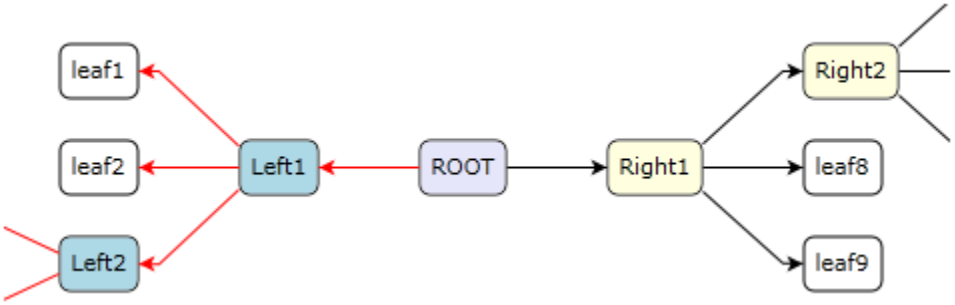
```
                        Converter={StaticResource theLinkBrushConverter}}"
              Points="8 4  0 8  2 4  0 0" go:LinkPanel.Index="-1"
              go:LinkPanel.Alignment="1 0.5" go:LinkPanel.Orientation="Along" />
  </go:LinkPanel>
</DataTemplate>
```

Note that in this example both the **Path.Stroke** and the **Polygon.Fill** are bound to the same data property using the same converter.

The **LinkBrushConverter** needs to convert the string value of **Info.LayoutId** to the desired **Brush**. This is an example of defining your own custom data converter:

```
public class LinkBrushConverter : Northwoods.GoXam.Converter {
  public override object Convert(object value, Type targetType,
              object parameter, System.Globalization.CultureInfo culture) {
    if (value is String) {
      switch ((String)value) {
        case "Right": return Black;
        case "Left": return Red;
        case "Up": return Green;
        case "Down": return Blue;
        default: return Black;
      }
    }
    return Black;
  }

  private static Brush Black = new SolidColorBrush(Colors.Black);
  private static Brush Red = new SolidColorBrush(Colors.Red);
  private static Brush Green = new SolidColorBrush(Colors.Green);
  private static Brush Blue = new SolidColorBrush(Colors.Blue);
}
```

For efficiency this example converter only returns one of four predefined solid brushes that are shared. However, it is common to return a `new SolidColorBrush` when the color is more variable. In any case, this is what the results might look like:



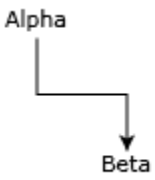
Link Routes

So far all of the example links have been fairly simple. If you want to customize the path that each link takes, you need to set properties on the link's **Route**. Each **Link** has a **Route** that it creates by default, but you can replace it with one that you have initialized.

```
<DataTemplate x:Key="LinkTemplate4">
```

```
  <go:LinkPanel go:Part.SelectionElementName="Path"
                go:Part.SelectionAdorned="True">
    <go:Link.Route>
      <go:Route Routing="Orthogonal" />
    </go:Link.Route>
    <go:LinkShape x:Name="Path" go:LinkPanel.IsLinkShape="True"
                  Stroke="Black" StrokeThickness="1" />
    <Polygon Fill="Black" Points="8 4  0 8  2 4  0 0"
             go:LinkPanel.Alignment="1 0.5" go:LinkPanel.Index="-1"
             go:LinkPanel.Orientation="Along" />
  </go:LinkPanel>
</DataTemplate>
```

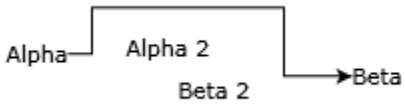
The **Route.Routing** property controls what general route the link will take. The default value is **LinkRouting.Normal**, which produces the direct paths you have seen so far. But if you use **LinkRouting.Orthogonal**, which tries to make each segment of the link either horizontal or vertical, it might look like:



Another routing option assumes orthogonal segments for the link, but also tries to avoid crossing over other nodes.

```
    <go:Link.Route>
      <go:Route Routing="AvoidsNodes" />
    </go:Link.Route>
```

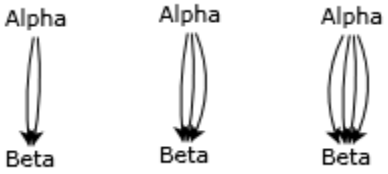
After adding two nodes to be in the way:



The **Route.Curve** property specifies what kind of path to draw given the points calculated for the route. The default value is **LinkCurve.None**, which produces the straight line segments you have seen in the examples so far. The **LinkCurve.Bezier** value produces naturally curved paths.

```
    <go:Link.Route>
      <go:Route Curve="Bezier" />
    </go:Link.Route>
```

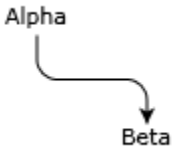
You can control the amount of curvature by setting the **Route.Curviness** property. With varying numbers of links between the same pair of nodes it will automatically compute values for **Curviness** unless you assign it explicitly.



Combining orthogonal **Routing** and **Corner**:

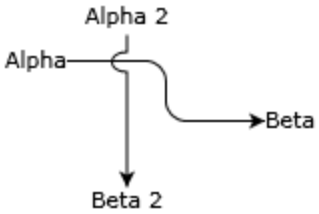
```
<go:Link.Route>
  <go:Route Routing="Orthogonal" Corner="10" />
</go:Link.Route>
```

produces:



Or use **Curve.JumpOver** with **LinkRouting.Orthogonal** or **AvoidsNodes**:

```
<go:Link.Route>
  <go:Route Routing="Orthogonal" Curve="JumpOver" Corner="10" />
</go:Link.Route>
```



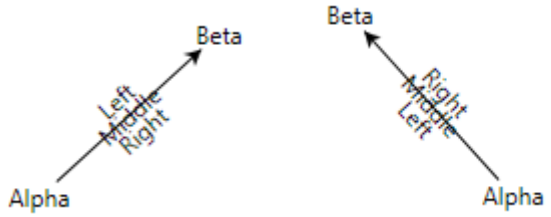
Link Labels

It is common to add annotations or decorations to links, particularly text. You can easily add any elements you want to a **LinkPanel**. For example, let us add three text labels to a link, one in the middle, one on the left side of the link and one on the right side of the link:

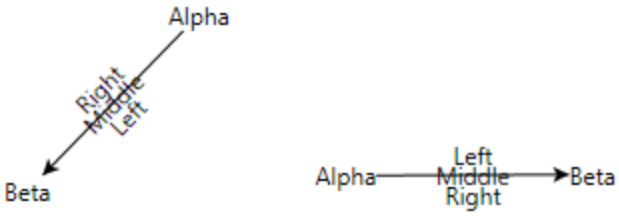
```
<DataTemplate x:Key="LinkTemplate5">
  <go:LinkPanel>
    <go:LinkShape Stroke="Black" StrokeThickness="1" />
    <Polygon Fill="Black" Points="8 4  0 8  2 4  0 0" go:LinkPanel.Index="-1"
             go:LinkPanel.Alignment="1 0.5" go:LinkPanel.Orientation="Along" />
    <TextBlock Text="Left"
               go:LinkPanel.Offset="0 -10" go:LinkPanel.Orientation="Upright" />
    <TextBlock Text="Middle"
               go:LinkPanel.Offset="0 0" go:LinkPanel.Orientation="Upright" />
    <TextBlock Text="Right"
               go:LinkPanel.Offset="0 10" go:LinkPanel.Orientation="Upright" />
  </go:LinkPanel>
</DataTemplate>
```

The **LinkPanel.Offset** attached property controls where to position the element relative to a point on a segment of the link. A positive value for the Y offset moves the label element towards the right side of the link, as seen going in the direction of the link. Naturally a negative value for the Y offset moves it towards the left side.

The segment is specified by the **LinkPanel.Index** attached property, which defaults to the middle of the whole link. The offset is rotated according to the angle formed by that link segment. Here are the results, with the nodes at different relative positions to demonstrate how the labels follow the (only) segment of the link.

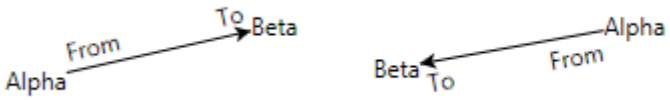


The **LinkPanel.Orientation** attached property controls the angle of the label relative to the angle of the link segment. The value of **Along**, as you have seen above with arrowheads, results in a label angle that is the same as the segment's angle. The value of **Upright** is useful for elements containing text because the text will not be upside down, although like **Along** it will always be angled to follow the link. To continue the counter-clockwise rotation of the Beta node around the Alpha node:



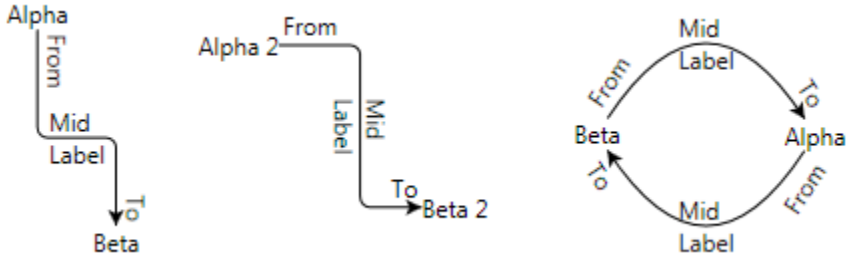
When you specify the **LinkPanel.Index**, you can position labels at places other than the middle of the link. The index of zero is at the very beginning of the link; a value of one is at the next point in the route. Negative values are permitted – they count down from the "to" end of the link, with index -1 at the very last point of the link.

```
<TextBlock Text="From" go:LinkPanel.Index="0"
           go:LinkPanel.Offset="NaN NaN" go:LinkPanel.Orientation="Upright" />
<TextBlock Text="To" go:LinkPanel.Index="-1"
           go:LinkPanel.Offset="NaN NaN" go:LinkPanel.Orientation="Upright" />
```



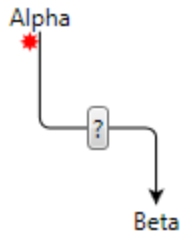
The uses of **NaN** in the **Offset** mean half the width and half the height of the label element, which is convenient when the size of the label element may vary.

Links need not be straight with a single segment. Here are examples of **Orthogonal** routing and of **Bezier** curves, with the middle label having two lines of text:



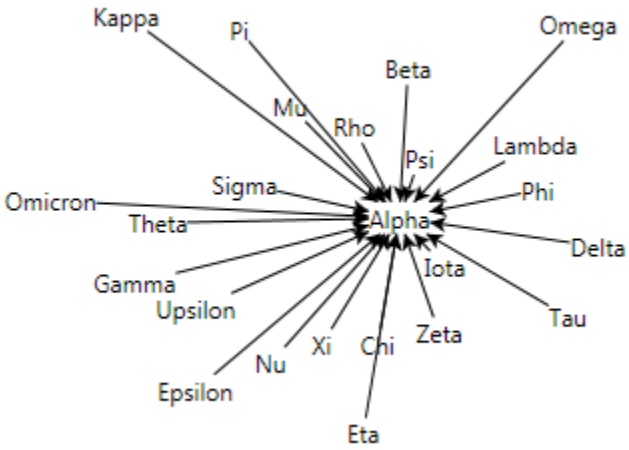
Labels need not be **TextBlock**s. The default **LinkPanel.Orientation** is **None**, meaning that the label element is not rotated at all. For example:

```
<!-- LinkPanel labels -->
<go:NodeShape go:LinkPanel.Index="0" go:LinkPanel.Offset="5 5"
              go:NodePanel.Figure="EightPointedStar" Fill="Red"
              Width="10" Height="10" />
<Button Content="?" Click="Button_Click" />
```



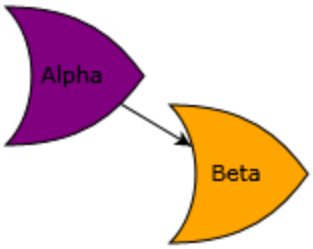
Link Connection Points on Nodes

In the examples above you have seen how each link will end at the edge of the node. To illustrate this further, notice in the following screenshot where the arrowheads appear to terminate around the "Alpha" node, around the rectangular bounds of the text:



If the node is not shaped like a rectangle, the link will connect at the edge.

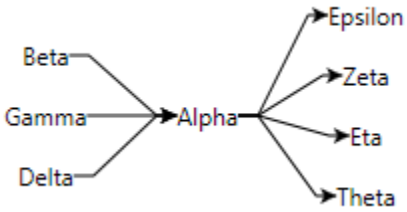
```
<DataTemplate x:Key="NodeTemplate1">
  <go:NodePanel go:Node.SelectionAdorned="True">
    <go:NodeShape go:NodePanel.Figure="OrGate" Width="70" Height="70"
          Stroke="Black" StrokeThickness="1"
          Fill="{Binding Path=Data.Color,
                         Converter={StaticResource theStringBrushConverter}}" />
    <TextBlock Text="{Binding Path=Data.Key}" TextAlignment="Center"
               HorizontalAlignment="Center" VerticalAlignment="Center" />
  </go:NodePanel>
</DataTemplate>
```



But what if you want to limit the points at which links may connect to a node? You can do so by setting the **Node.FromSpot** and **Node.ToSpot** attached properties on the root visual element of the node. The default value is **Spot.None**, which means to calculate a point along the edge of the element. But you can specify spot values that describe particular positions on the element. For example:

```
<DataTemplate x:Key="NodeTemplate2">
  <TextBlock Text="{Binding Path=Data.Key}" go:Node.SelectionAdorned="True"
             go:Node.ToSpot="MiddleLeft" go:Node.FromSpot="MiddleRight" />
</DataTemplate>
```

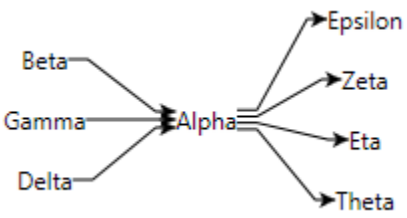
This specifies that links coming into this node connect at the middle of the left side, and that links going out of this node connect at the middle of the right side. Such a convention is appropriate for diagrams that have a general sense of direction to them, such as the following one which goes from left to right:



You can also specify that the links go into a node not at a single spot but spread out along one side. Change the previous example to use:

```
    go:Node.ToSpot="LeftSide" go:Node.FromSpot="RightSide"
```

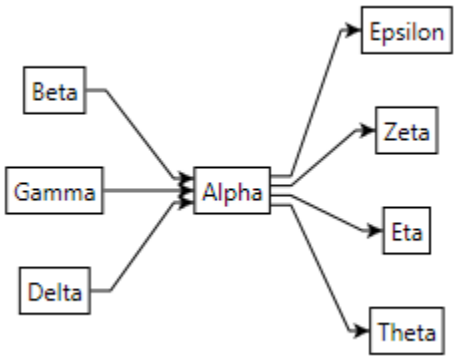
And you will get:



Of course specifying a side works well only for nodes that are basically rectangular and probably larger than in this case. So let's add a border around the text to make each node bigger:

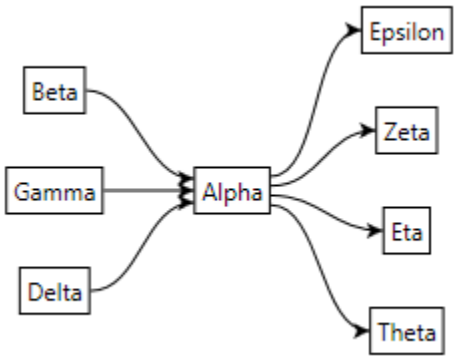
```
<DataTemplate x:Key="NodeTemplate3">
  <Border BorderBrush="Black" BorderThickness="1" Padding="3"
          go:Node.SelectionAdorned="True"
          go:Node.ToSpot="LeftSide" go:Node.FromSpot="RightSide" >
    <TextBlock Text="{Binding Path=Data.Key}" />
  </Border>
</DataTemplate>
```

Note how the attached node properties have been moved to the new root element of the data template. This node template with the same data results in:



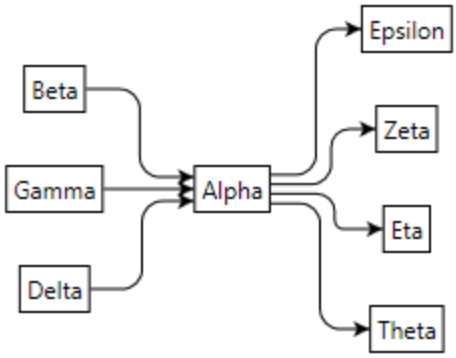
Of course you can use different kinds of **Route**s for the link template. Consider:

```
<go:Link.Route>
  <go:Route Curve="Bezier" />
</go:Link.Route>
```



Or:

```
<go:Link.Route>
  <go:Route Routing="Orthogonal" Corner="10" />
</go:Link.Route>
```



Ports on Nodes

Although you have some control over where links will connect at a node (at a particular spot, along one or more sides, or at the intersection with the edge), there are times when you want to have different logical and graphical places at which links should connect. The elements to which a link may connect are called *ports*. There may be any number of ports in a node. By default there is just one port, the root visual element, which results in the effect of having the whole node act as the port, as you have seen above. Support for multiple ports is only possible in a **GraphLinksModel** because only when you have separate data for each link can you attach information describing which port the link should connect to.

To declare that a particular element is a port, set the **Node.PortId** attached property on it. Unlike most of the **Part** and **Node** attached properties, which may only be applied to the root visual element of the node, the port-related **Node** attached properties may apply to any element in the visual tree of the node. These attached properties have names that start with "Port", "From", "To", or "Linkable".

```
<DataTemplate x:Key="NodeTemplate4">
  <Border BorderBrush="Black" BorderThickness="1"
          go:Node.SelectionAdorned="True">
    <Grid Background="LightGray">
      <Grid.ColumnDefinitions>
        <ColumnDefinition Width="Auto" />
        <ColumnDefinition Width="*" />
        <ColumnDefinition Width="Auto" />
      </Grid.ColumnDefinitions>
      <Grid.RowDefinitions>
        <RowDefinition Height="Auto" />
        <RowDefinition Height="*" />
        <RowDefinition Height="*" />
      </Grid.RowDefinitions>
      <TextBlock Grid.Column="0" Grid.Row="0" Grid.ColumnSpan="3"
                 Text="{Binding Path=Data.Key}" TextAlignment="Center"
                 FontWeight="Bold" TextWrapping="Wrap" Margin="4,4,4,2" />
      <StackPanel Grid.Column="0" Grid.Row="1" Orientation="Horizontal">
        <!-- this Rectangle is a port, identified with the string "A";
             links only come into it at the middle of the left side -->
```

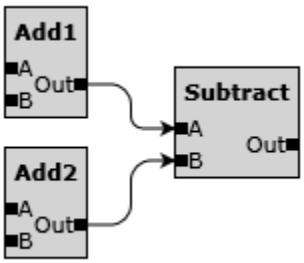
```
        <Rectangle Width="6" Height="6" Fill="Black"
                   go:Node.PortId="A" go:Node.ToSpot="MiddleLeft" />
        <TextBlock Text="A" />
      </StackPanel>
      <StackPanel Grid.Column="0" Grid.Row="2" Orientation="Horizontal">
        <!-- this Rectangle is another input port, named "B" -->
        <Rectangle Width="6" Height="6" Fill="Black"
                   go:Node.PortId="B" go:Node.ToSpot="MiddleLeft" />
        <TextBlock Text="B" />
      </StackPanel>
      <StackPanel Grid.Column="2" Grid.Row="1" Grid.RowSpan="2"
                  Orientation="Horizontal" VerticalAlignment="Center">
        <TextBlock Text="Out" />
        <!-- this Rectangle is another port, identified with the string "Out";
             links only go out of it at the middle of the right side -->
        <Rectangle Width="6" Height="6" Fill="Black"
                   go:Node.PortId="Out" go:Node.FromSpot="MiddleRight" />
      </StackPanel>
    </Grid>
  </Border>
</DataTemplate>
```

Each port has a **Node.PortId** that corresponds to the optional port parameter information at both ends of each link. To avoid visual confusion in this example there is also a **TextBlock** next to each port, showing the same string.

This node template, combined with a **GraphLinksModel** and data such as:

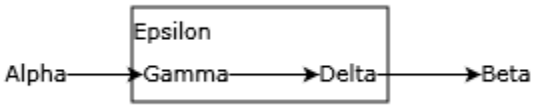
```
var model = new GraphLinksModel<MyData, String, String, MyLinkData>();

model.NodesSource = new ObservableCollection<MyData>() {
  new MyData() { Key="Add1" },
  new MyData() { Key="Add2"},
  new MyData() { Key="Subtract" },
};

model.LinksSource = new ObservableCollection<MyLinkData>() {
  new MyLinkData() { From="Add1", FromPort="Out", To="Subtract", ToPort="A" },
  new MyLinkData() { From="Add2", FromPort="Out", To="Subtract", ToPort="B" },
};

myDiagram.Model = model;
```

can produce a diagram like:



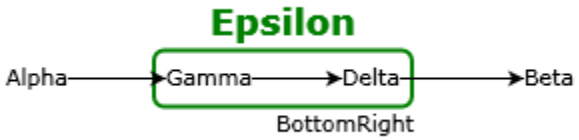
Data Templates for Groups

To define the appearance of group nodes, you can set the **Diagram.GroupTemplate** property. The default template produces the following simple representation of a group, in this case "Epsilon".



To customize the appearance of a group, you could define a template such as:

```
<DataTemplate x:Key="GroupTemplate1">
  <StackPanel go:Node.LocationElementName="myGroupPanel">
    <!-- This is the "header" for the group -->
    <TextBlock x:Name="Label" Text="{Binding Path=Data.Key}"
               FontSize="18" FontWeight="Bold" Foreground="Green"
               HorizontalAlignment="Center"/>
    <Border x:Name="myBorder" CornerRadius="5"
            BorderBrush="Green" BorderThickness="2">
      <!-- The GroupPanel is the placeholder for member parts -->
      <go:GroupPanel x:Name="myGroupPanel" Padding="5" />
    </Border>
    <!-- This is some extra information for the group -->
    <TextBlock Text="BottomRight" HorizontalAlignment="Right" />
  </StackPanel>
</DataTemplate>
```



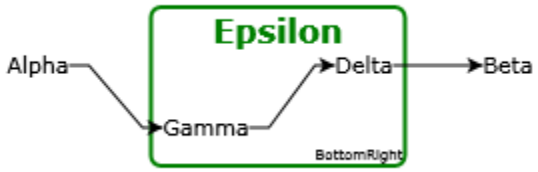
Notice that there is a **GroupPanel** element inside the **Border**. You use a **GroupPanel** as the placeholder for all of the nodes and links that are members of the group. The member **Node**s and **Link**s are not visual children of the panel or of the group node – they are independent parts in the diagram.

If you use a **GroupPanel**, and if it is not the root visual element of the data template, it must be named as the **Node.LocationElementName** for the group. Just give the **GroupPanel** a **Name** and refer to it via the attached property **Node.LocationElementName** on the root element. This means that the **Node**'s location will always be the same as the **GroupPanel**'s location, even as elements outside of the **GroupPanel** change size or move around with respect to the panel.

```
<DataTemplate x:Key="GroupTemplate2">
  <Border x:Name="myBorder" CornerRadius="5"
          BorderBrush="Green" BorderThickness="2"
          go:Node.LocationElementName="myGroupPanel">
```

```
    <StackPanel>
      <TextBlock x:Name="Label" Text="{Binding Path=Data.Key}"
                 FontSize="16" FontWeight="Bold" Foreground="Green"
                 HorizontalAlignment="Center" />
      <go:GroupPanel x:Name="myGroupPanel" Padding="5" />
      <TextBlock Text="BottomRight" FontSize="7"
                 HorizontalAlignment="Right" />
    </StackPanel>
  </Border>
</DataTemplate>
```

The second screenshot shows the result of dragging the “Gamma” node downward a bit.



A **GroupPanel** always encloses its member **Node**s, even while the nodes are being dragged. If you don’t want this behavior during dragging, for example in order to permit a **Node** to be dragged outside of its **Group**, you can set **GroupPanel.SurroundsMembersAfterDrop** to true. This changes the behavior of the **GroupPanel** so that it does not resize during a drag until the drop is completed.

Collapsing and Expanding SubGraphs

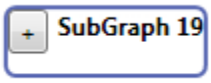
It is common to simplify graphs by collapsing subgraphs into a single node. One way to implement collapsible subgraphs is with a button.

```
<!-- show either a "+" or a "-" as the Button content -->
<go:BooleanStringConverter x:Key="theButtonConverter"
                           TrueString="-" FalseString="+" />

<DataTemplate x:Key="GroupTemplate">
  <Border CornerRadius="5" BorderThickness="2" Background="Transparent"
          BorderBrush="{Binding Path=Data.Color,
                                Converter={StaticResource theBrushConverter}}"
          go:Part.SelectionAdorned="True"
          go:Node.LocationElementName="myGroupPanel"
          go:Group.IsSubGraphExpanded="False">
    <!-- go:Group.IsSubGraphExpanded="False" causes it to start collapsed -->
    <StackPanel>
      <StackPanel Orientation="Horizontal" HorizontalAlignment="Left">
        <Button x:Name="myCollapseExpandButton"
            Click="CollapseExpandButton_Click"
```

```
            Content="{Binding Path=Group.IsExpandedSubGraph,
                              Converter={StaticResource theButtonConverter}}"
            Width="20" Margin="0 0 5 0"/>
          <TextBlock Text="{Binding Path=Data.Key}" FontWeight="Bold" />
        </StackPanel>
        <go:GroupPanel x:Name="myGroupPanel" Padding="5" />
      </StackPanel>
      <!-- each Group can have its own Layout -->
      <go:Group.Layout>
        <!-- this Layout is performed whenever any nested Group changes size -->
        <go:LayeredDigraphLayout Direction="90"
                                 Conditions="Standard GroupSizeChanged" />
      </go:Group.Layout>
    </Border>
</DataTemplate>
```

Note that the **Button.Content** is bound to the **Group.IsExpandedSubGraph** property, via a converter that converts the boolean value to either the string “+” or the string “-“.

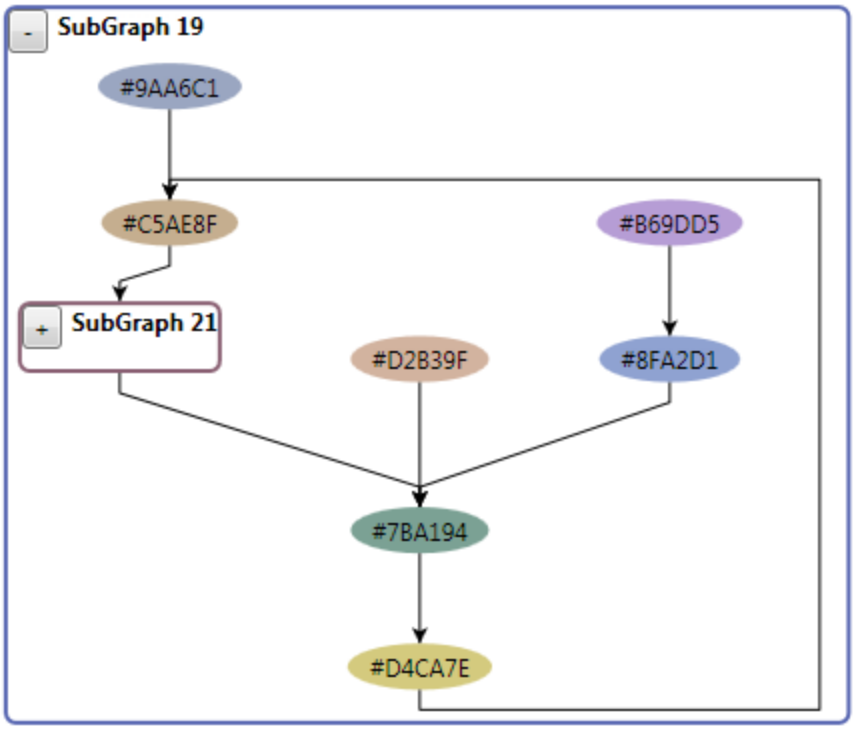
Collapsed it might appear as:



The **Button.Click** event handler might be defined as:

```
private void CollapseExpandButton_Click(object sender, RoutedEventArgs e) {
  // the Button is in the visual tree of a Node
  Button button = (Button)sender;
  Group sg = Part.FindAncestor<Group>(button);
  if (sg != null) {
    SimpleData subgraphdata = (SimpleData)sg.Data;
    // always make changes within a transaction
    myDiagram.StartTransaction("CollapseExpand");
    // toggle whether this node is expanded or collapsed
    sg.IsExpandedSubGraph = !sg.IsExpandedSubGraph;
    myDiagram.CommitTransaction("CollapseExpand");
  }
}
```

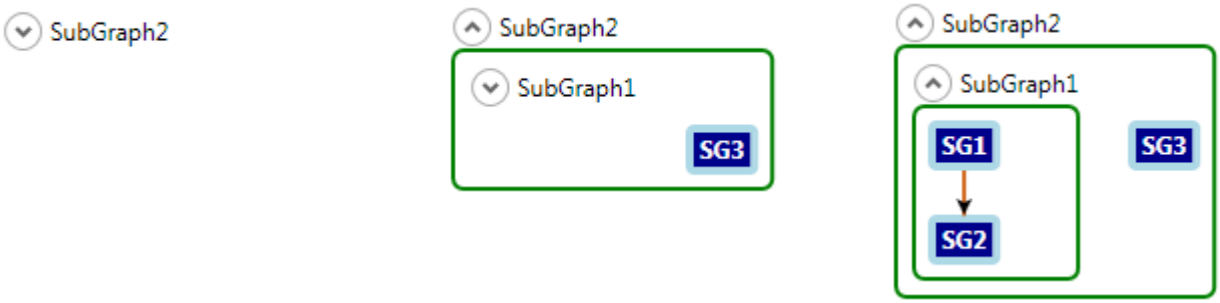
Expanded it might look like:



A more “standard” implementation for a **Group** might use an **Expander**:

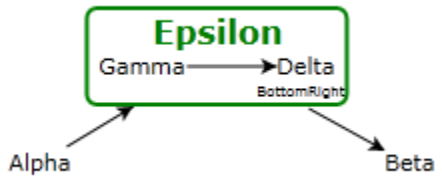
```
<DataTemplate x:Key="GroupTemplate6">
  <Expander Header="{Binding Path=Data.Name}"
            IsExpanded="{Binding Path=Group.IsExpandedSubGraph, Mode=TwoWay}"
            go:Node.LocationElementName="myGroupPanel">
    <Border BorderBrush="Green" BorderThickness="2"
            Background="Transparent" CornerRadius="5">
      <go:GroupPanel x:Name="myGroupPanel" Padding="6" />
    </Border>
  </Expander>
</DataTemplate>
```

Note how the **Expander.IsExpanded** property is data-bound to **Group.IsExpandedSubGraph**.



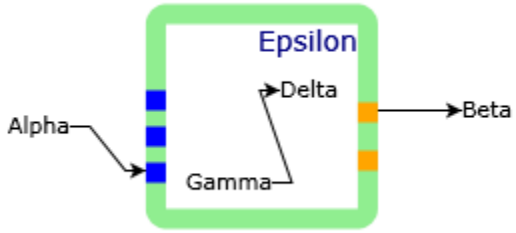
Groups with Ports

The previous examples did not treat groups as nodes in their own right. As with regular **Node**s, a link to a **Group** will by default treat the whole node as the only "port". For example, connecting "Alpha" to "Epsilon" (instead of to "Gamma") and "Epsilon" (instead of "Delta") to "Beta" might result in the following screenshot. The "Alpha" and "Beta" nodes have been moved to make clearer the connections to the group.



The following example gives group nodes three ports on the left and two on the right, spaced equally within the thick border. The input ports on the left are named "zero", "one", and "two"; the output ports on the right are named "OutA" and "OutB". This example has no text labels to visually name each port.

```
<DataTemplate x:Key="GroupTemplate3">
  <Border x:Name="myBorder" CornerRadius="5"
          BorderBrush="LightGreen" BorderThickness="10"
          go:Node.LocationElementName="myGroupPanel">
    <go:GroupPanel x:Name="myGroupPanel" Padding="10 5 10 5" Margin="0 20 0 0" >
      <TextBlock x:Name="Label" go:Node.PortId=""
                 Text="{Binding Path=Data.Key}" FontSize="14" Foreground="Navy"
                 go:SpotPanel.Spot="1 0 0 -2" go:SpotPanel.Alignment="1 1" />
      <Rectangle go:SpotPanel.Spot="0 0.25" go:SpotPanel.Alignment="1 0.5"
                 Fill="Blue" Width="10" Height="10" go:Node.PortId="zero" />
      <Rectangle go:SpotPanel.Spot="0 0.50" go:SpotPanel.Alignment="1 0.5"
                 Fill="Blue" Width="10" Height="10" go:Node.PortId="one" />
      <Rectangle go:SpotPanel.Spot="0 0.75" go:SpotPanel.Alignment="1 0.5"
                 Fill="Blue" Width="10" Height="10" go:Node.PortId="two" />
      <Rectangle go:SpotPanel.Spot="1 0.33" go:SpotPanel.Alignment="0 0.5"
                 Fill="Orange" Width="10" Height="10" go:Node.PortId="OutA" />
      <Rectangle go:SpotPanel.Spot="1 0.67" go:SpotPanel.Alignment="0 0.5"
                 Fill="Orange" Width="10" Height="10" go:Node.PortId="OutB" />
    </go:GroupPanel>
  </Border>
</DataTemplate>
```



This diagram was created with the same node data as before but with the following link data:

```
model.LinksSource = new ObservableCollection<MyLinkData>() {
  new MyLinkData() { From="Alpha", To="Epsilon", ToPort="two" },
  new MyLinkData() { From="Gamma", To="Delta" },
  new MyLinkData() { From="Epsilon", To="Beta", FromPort="OutA" },
};
```

Groups as Independent Containers

The above examples all intend to have each group exactly surround its collection of member nodes plus some padding. However, there are other scenarios where you want to treat each group as a fixed size box where the user might add or remove items (i.e. nodes) via drag-and-drop.

```
<DataTemplate x:Key="GroupTemplateFixedSize">
  <StackPanel go:Node.LocationElementName="main"
              go:Part.SelectionElementName="main"
              go:Part.SelectionAdorned="True"
              go:Part.DropOntoBehavior="AddsToGroup">
    <TextBlock Text="{Binding Path=Data.Key}" FontWeight="Bold"
               HorizontalAlignment="Left" />
    <Rectangle x:Name="main" Fill="White" StrokeThickness="3"
               Stroke="{Binding Path=Part.IsDropOntoAccepted,
                                Converter={StaticResource theStrokeChooser}}"
               Width="100" Height="100" />
  </StackPanel>
</DataTemplate>
```

Note the addition of `go:Part.DropOntoBehavior="AddsToGroup"`. You can enable “drop onto” behavior by adding this attached property on groups and by also setting **DraggingTool.DropOntoEnabled** to true:

```
<go:Diagram Grid.Row="0" . . . >
  <go:Diagram.DraggingTool>
    <go:DraggingTool DropOntoEnabled="True" />
  </go:Diagram.DraggingTool>
</go:Diagram>
```

This will allow users to drag nodes into and out of this rectangular box. When the drop occurs, the nodes become members of the group. That means that copying the group will also copy the members, and that deleting the group will also delete the members. Dragging a node out of such a group also removes it from that group – copying or deleting the group will have no effect on the dragged node.

To help provide feedback to the user, note the binding of the **Rectangle.Stroke** on the **Part.IsDropOntoAccepted** property. The **DraggingTool** will temporarily set that **Part** property during the dragging process if the dragged nodes might be added to that **Group**. You can override the **DraggingTool.IsValidMember** predicate to return false if you do not want a particular node to become a member of a particular group. For example, in the Planogram sample, **IsValidMember** is

defined to return false when the dragged node is a Rack or a Shelf, to prevent nesting of Racks or Shelves.

The Planogram sample also demonstrates how these groups can be resizable by the user. Because the template is not using a **GroupPanel**, there are no inherent limits on where the group appears to be relative to its member nodes.

However, there may be times when you want to use a **GroupPanel** most of the time, but you still want to support drag-and-drop re-parenting of nodes between groups. The problem with the use of a **GroupPanel** is that as the user tries to drag a member node out of a group, the group automatically expands to include its member node. In this particular case you can use a **GroupPanel** when you also set its **SurroundsMembersAfterDrop** property to true. Basically the auto-sizing behavior of a **GroupPanel** is temporarily disabled during a move conducted by the **DraggingTool**.

```
<DataTemplate x:Key="GroupTemplateAddableRemovable">
  <StackPanel go:Node.LocationElementName="main"
              go:Part.SelectionElementName="main"
              go:Part.SelectionAdorned="True"
              go:Part.DropOntoBehavior="AddsToGroup">
    <TextBlock Text="{Binding Path=Data.Key}" FontWeight="Bold"
               HorizontalAlignment="Left" />
    <Border Background="White" BorderThickness="3" CornerRadius="5"
            BorderBrush="{Binding Path=Part.IsDropOntoAccepted,
                                  Converter={StaticResource theStrokeChooser}}">
      <go:GroupPanel x:Name="main" SurroundsMembersAfterDrop="True"
                     MinWidth="100" MinHeight="100" />
    </Border>
  </StackPanel>
</DataTemplate>
```

Layout

The positioning of **FrameworkElement**s in **Node**s is achieved with the standard WPF layout system, primarily the use of various kinds of **Panel**s.

In GoXam diagrams, you can position a node by setting or data-binding in XAML the **Node.Location** attached property on its root visual element, or by setting programmatically the **Node.Location** property. And users can reposition a node by dragging it.

However, there are also some automated means of positioning the nodes. These are implemented by several **DiagramLayout** classes, primarily: **GridLayout**, **CircularLayout**, **TreeLayout**, **ForceDirectedLayout**, and **LayeredDigraphLayout**. Any layout can work with any kind of model.

A layout can be associated with a whole diagram by setting the **Diagram.Layout** property.

```
<go:Diagram . . .>
  <go:Diagram.Layout>
    <go:TreeLayout . . . />
  </go:Diagram.Layout>
</go:Diagram>
```

A layout can also be associated with a **Group** by setting the **Group.Layout** attached property. If a **Group** has a layout, that layout will only position the members (nodes and links) of the group, and the **Diagram**'s layout will not operate on those members but will treat the group as a single node.

Because there may be many layouts present in a diagram, the **Diagram.LayoutManager** is responsible for managing them, including deciding when they need to run again. By default there are a number of events that may cause a re-layout. These cases are specified by the **LayoutChange** enumeration, such as **LayoutChange .NodeAdded** or **LayoutChange .LinkRemoved**.

Each **DiagramLayout** has a **Conditions** property that governs which **LayoutChange**s will cause a re-layout. The default behavior is to perform another layout when any node, link, or group membership is added or removed, or when a **Layout** is replaced or when a template is replaced. If you don't want a layout to happen when users delete nodes or links, you could say:

```
    <go:TreeLayout Conditions="NodeAdded LinkAdded" . . . />
```

Then only when the user adds a node or draws a new link (or reconnects an existing one) will a layout automatically occur.

The most commonly set properties on **LayoutManager** involve animation. By default the **LayoutManager.Animated** property is true, so that each layout will cause top-level nodes to move smoothly from their original location to their new one. (Nodes that are members of groups will move instantly.) The default animation time is 500 milliseconds.

```
<go:Diagram . . .>
  <go:Diagram.LayoutManager>
    <go:LayoutManager AnimationTime="1000" />
  </go:Diagram.LayoutManager>
  <go:Diagram.Layout>
    <go:TreeLayout . . . />
  </go:Diagram.Layout>
</go:Diagram>
```

Normally all of the nodes and links in the diagram are laid out by the **Diagram.Layout**. You can cause a node or link not to participate in a layout by setting its **Part.LayoutId** property to "None" on the root element of the node or link template:

```
    go:Part.LayoutId="None"
```

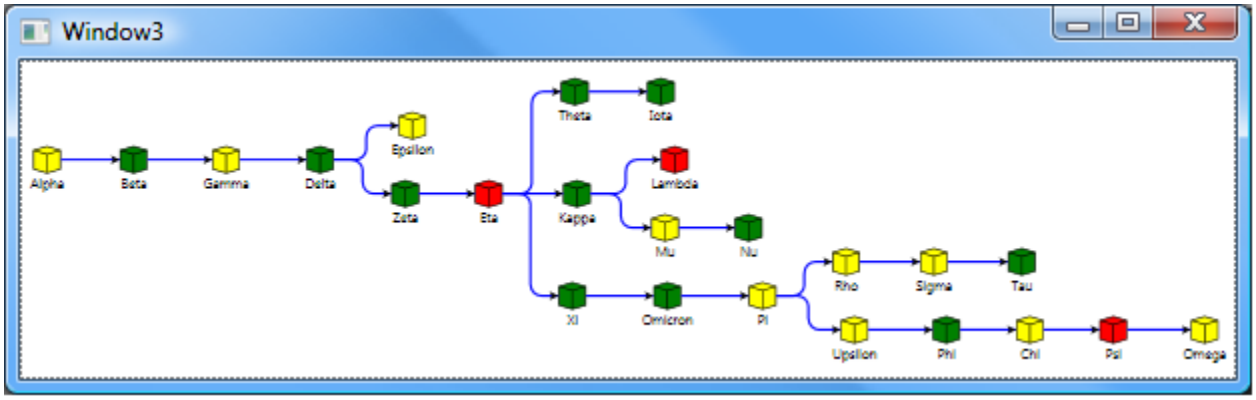
Nodes that are not laid out will not be positioned; links that are not laid out will not be routed specially and will not be considered when arranging the connected nodes.

TreeLayout

The simplest layout involves tree structures. It is very fast and can handle many nodes.

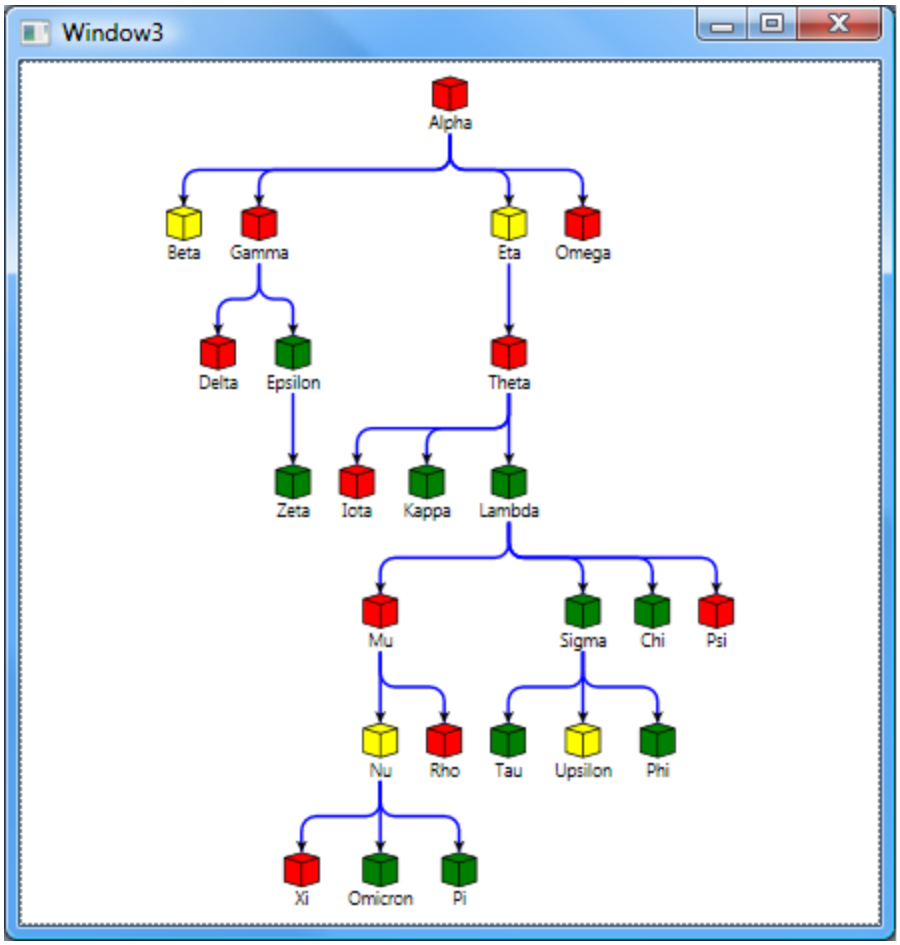
```
<go:Diagram . . .>
  <go:Diagram.Layout>
    <go:TreeLayout />
  </go:Diagram.Layout>
</go:Diagram>
```

With a model containing node data forming a tree structure, the result might look like:



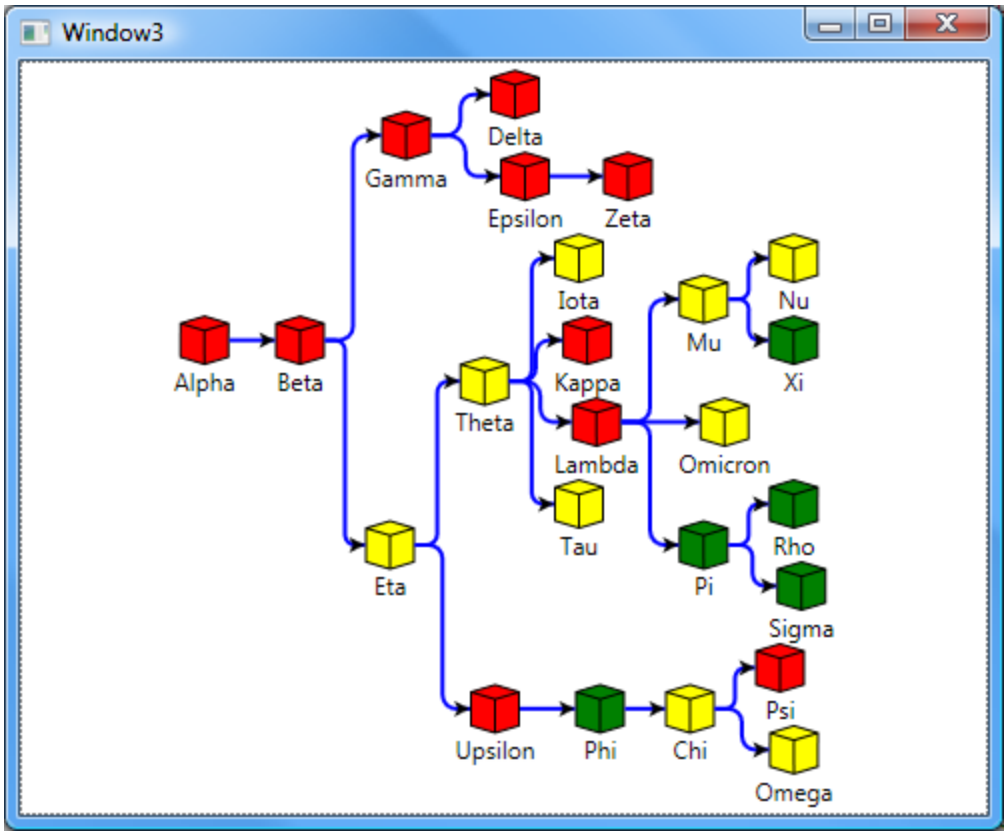
There are is a lot of customization possible for trees. **Angle** controls the general growth direction – it must be 0 (towards the right), 90 (downward), 180 (leftward) or 270 (upward). **Alignment** controls how the parent node is positioned relative to its children.

```
  <go:TreeLayout Angle="90" Alignment="CenterSubtrees" />
```



You can control how closely the layers and the nodes are placed. For example, you can really pack them close together with:

```
<go:TreeLayout LayerSpacing="20" NodeSpacing="0" />
```



You can have the children of each node be sorted. By default the **TreeLayout.Comparer** compares the **Node.Text** property. So if the **Diagram.NodeTemplate** includes:

```
go:Part.Text="{Binding Path=Data.Key}"
```

on the root element, and if you specify the **TreeLayout.Sorting** property:

```
<go:TreeLayout Angle="90" Alignment="Start" Sorting="Ascending" />
```

The set of children for each node is alphabetized. (In this case that means alphabetical ordering of the English names of the letters of the Greek alphabet.)

If your graph structure is mostly tree-like, but you have a few “extra” links that should be ignored for the purpose of deciding the tree structure, you can set the **Part.LayoutId** attached property on those links to be “None”.

You can experiment with the **TreeLayout** properties in the TLayout sample of the demo.

ForceDirectedLayout

The **ForceDirectedLayout** uses forces similar to physical forces to push and pull nodes. Links are treated as if they were springs of a particular length and stiffness. Each node has an electrical charge that repels other nodes.

An example of a **ForceDirectedLayout**:

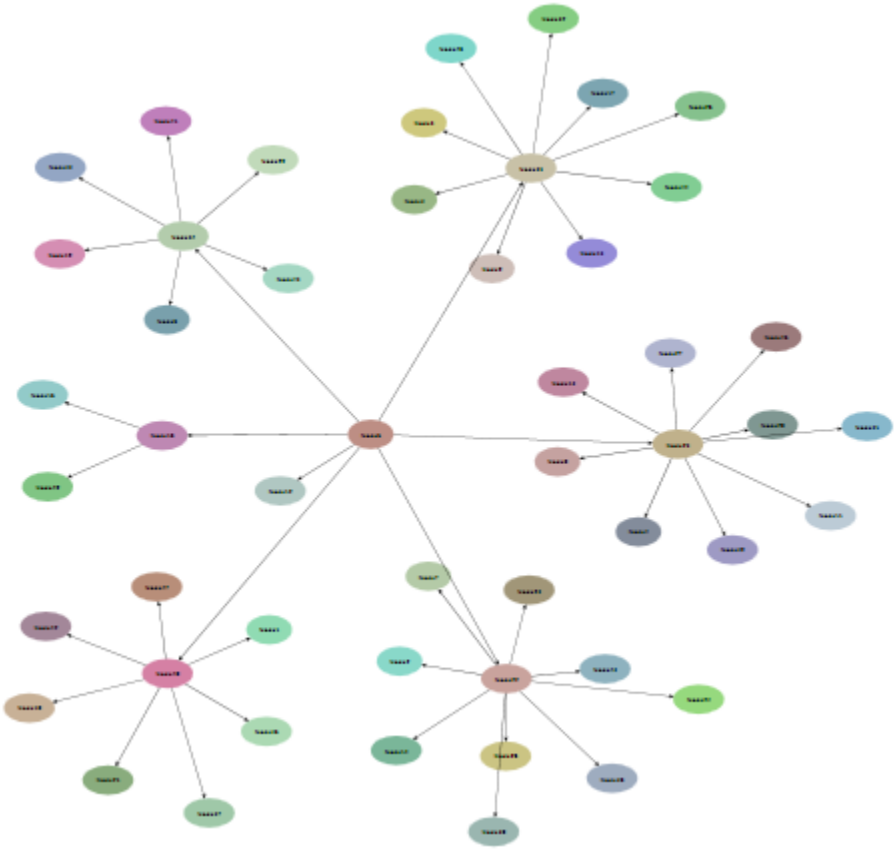
```
<go:ForceDirectedLayout DefaultSpringLength="10" DefaultElectricalCharge="50" />
```

For small nodes that do not have too much connectivity you can use smaller values than the defaults of 50 for the spring length and 150 for the electrical charge.

Unlike the other layouts, **ForceDirectedLayout** produces incremental results, so running it for longer (i.e. values of **ForceDirectedLayout.MaxIterations** > 100) may improve the results.

There are a number of properties that control the behavior of the layout. The ones most commonly set include **Conditions** and the **Default…** properties.

You can experiment with the **ForceDirectedLayout** properties in the FDLayout sample of the demo.



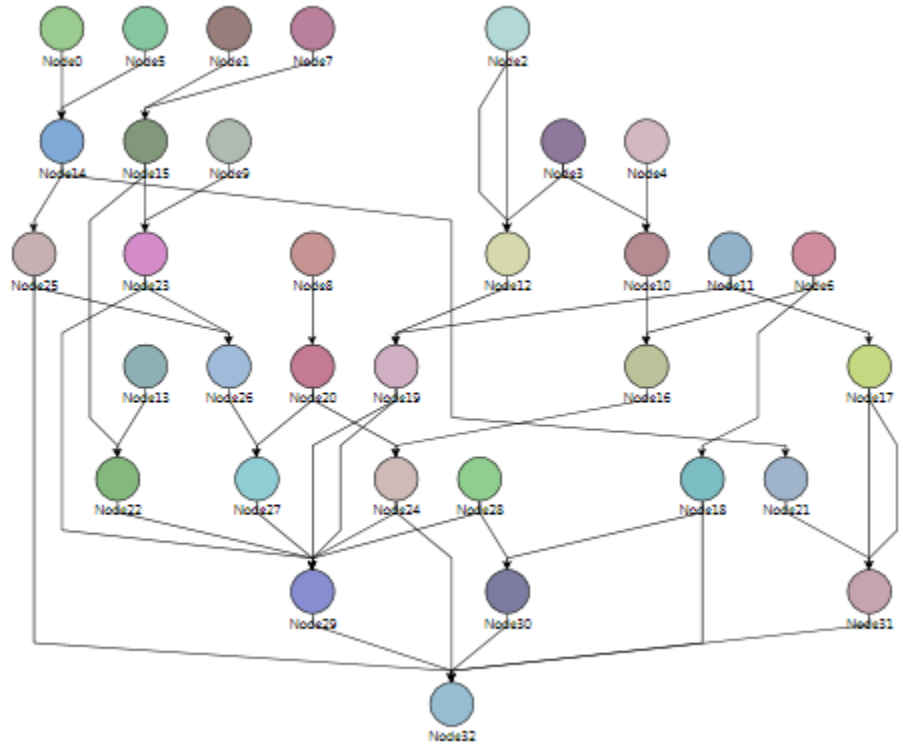
LayeredDigraphLayout

When the nodes of a graph can be naturally organized into layers but the structure is not tree-like, you can use **LayeredDigraphLayout**.

This layout can handle multiple links coming into a node as well as links that create cycles. However, it is slower than **TreeLayout**, and it does not have tree-specific customization features.

As with the other layouts, there are a number of properties that control its behavior. The ones most commonly set include **Direction**, **LayerSpacing**, **ColumnSpacing**, and **Conditions**.

You can experiment with the **LayeredDigraphLayout** properties in the LDLayout sample of the demo.

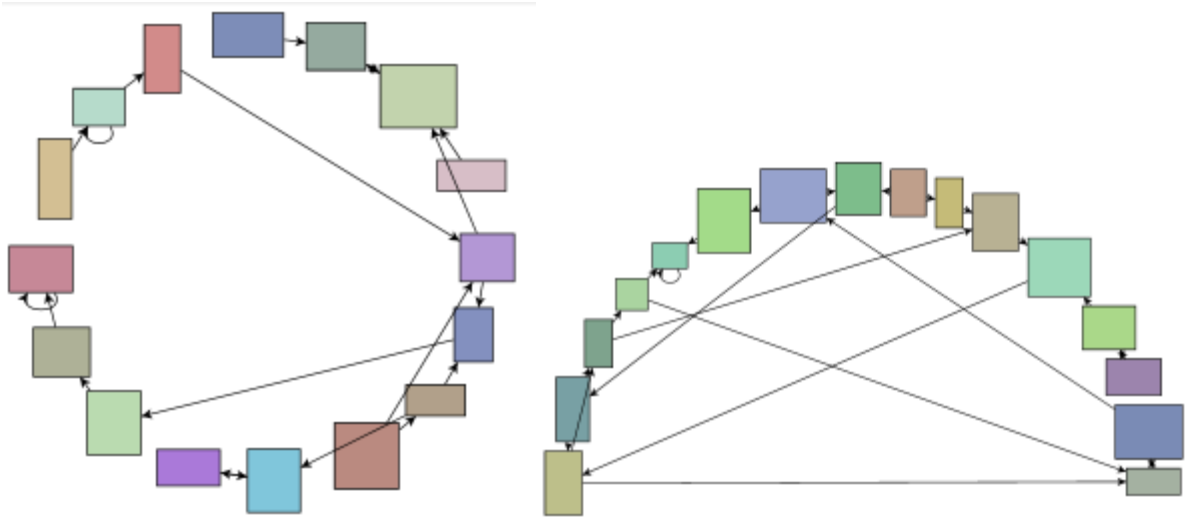


CircularLayout

The **CircularLayout** positions all of its nodes in a circular or elliptical pattern.

There are a number of properties that control the behavior of the layout. These include how the nodes are ordered, how they are spaced, the X radius of the ellipse, the aspect ratio of the ellipse, and the start and sweep angles of the ellipse that are occupied.

You can experiment with the **CircularLayout** properties in the CLayout sample of the demo.



Selection

Users can typically select and deselect parts by clicking on them or by clicking in the background. You can programmatically select or deselect a **Part** by setting its **Part.IsSelected** property.

The **Diagram** keeps a collection of selected parts, **Diagram.SelectedParts**. It also has a reference to the primary selected part: **Diagram.SelectedPart.** In order to show detail information about the primary selection it is natural to bind to **Diagram.SelectedPart**. If you only want to bind to the primary selection when it is a **Node** (and not a **Group**), bind to **Diagram.SelectedNode**. Similarly, you can bind to **Diagram.SelectedGroup** or **Diagram.SelectedLink.**

You can limit how many parts are selected by setting **Diagram.MaximumSelectionCount**.

You can show that a part is selected using either or both of two general techniques: adding **Adornment**s or changing the appearance of some of the elements of the visual tree.

Selection Adornments

It is common to display that a part is selected by having it show a selection **Adornment** when the part is selected. That is accomplished by setting the **Part.SelectionAdorned** attached property to true:

```
<DataTemplate x:Key="NodeTemplate1">
  <Grid go:Node.SelectionAdorned="True" . . .>
    . . .
  </Grid >
</DataTemplate>
```

This is the default selection adornment template, which defines what is shown when the part becomes selected:

```
<DataTemplate>
  <go:SelectionHandle Stroke="{x:Static SystemColors.HighlightBrush}"
```

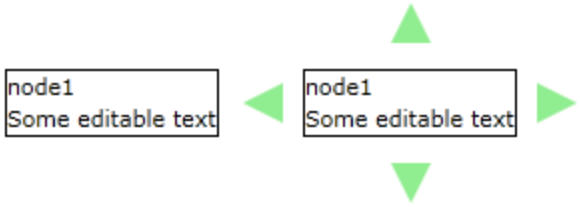
```
    StrokeThickness="3" go:Part.Selectable="False" SnapsToDevicePixels="True" />
</DataTemplate>
```

These **Adornment** shapes automatically take the shape of the **FrameworkElement** that is the selected part's **SelectionElement**.

But you can customize the elements that are shown when a part is selected by specifying its **SelectionAdornmentTemplate**. For example, you can arrange four triangles to be positioned outside of the adorned element by using a **Grid** with a **SpotPanel** in the middle cell:

```
<DataTemplate x:Key="OuterSelectionAdornmentTemplate">
  <Grid go:Node.LocationElementName="Main">
    <Grid.ColumnDefinitions>
      <ColumnDefinition Width="Auto" />
      <ColumnDefinition Width="Auto" />
      <ColumnDefinition Width="Auto" />
    </Grid.ColumnDefinitions>
    <Grid.RowDefinitions>
      <RowDefinition Height="Auto" />
      <RowDefinition Height="Auto" />
      <RowDefinition Height="Auto" />
    </Grid.RowDefinitions>
    <!-- when in an Adornment, automatically sized to the AdornedElement -->
    <go:SpotPanel Grid.Row="1" Grid.Column="1" x:Name="Main"
                  HorizontalAlignment="Center" VerticalAlignment="Center" />
    <!-- demonstrate triangles around the AdornedElement -->
    <go:NodeShape Grid.Row="0" Grid.Column="1" Margin="10"
        HorizontalAlignment="Center" VerticalAlignment="Center"
        Fill="LightGreen" go:NodePanel.Figure="TriangleUp"
        Width="20" Height="20" />
    <go:NodeShape Grid.Row="1" Grid.Column="0" Margin="10"
        HorizontalAlignment="Center" VerticalAlignment="Center"
        Fill="LightGreen" go:NodePanel.Figure="TriangleLeft"
        Width="20" Height="20" />
    <go:NodeShape Grid.Row="1" Grid.Column="2" Margin="10"
        HorizontalAlignment="Center" VerticalAlignment="Center"
        Fill="LightGreen" go:NodePanel.Figure="TriangleRight"
        Width="20" Height="20" />
    <go:NodeShape Grid.Row="2" Grid.Column="1" Margin="10"
        HorizontalAlignment="Center" VerticalAlignment="Center"
        Fill="LightGreen" go:NodePanel.Figure="TriangleDown"
        Width="20" Height="20" />
  </Grid>
</DataTemplate>
```

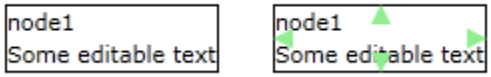
Here's what you might see with a node, both unselected and selected using this adornment template:



If you want to display some elements within the bounds, more or less, of the adorned element, you can use a **SpotPanel** in your adornment template:

```
<DataTemplate x:Key="InnerSelectionAdornmentTemplate">
  <!-- automatically sized to the AdornedElement -->
  <go:SpotPanel Grid.Row="1" Grid.Column="1"
                HorizontalAlignment="Center" VerticalAlignment="Center">
    <!-- demonstrate triangles just inside the AdornedElement -->
    <go:NodeShape go:SpotPanel.Spot="MiddleTop"
                  go:SpotPanel.Alignment="MiddleTop"
                  Fill="LightGreen" go:NodePanel.Figure="TriangleUp"
                  Width="10" Height="10" />
    <go:NodeShape go:SpotPanel.Spot="MiddleLeft"
                  go:SpotPanel.Alignment="MiddleLeft"
                  Fill="LightGreen" go:NodePanel.Figure="TriangleLeft"
                  Width="10" Height="10" />
    <go:NodeShape go:SpotPanel.Spot="MiddleRight"
                  go:SpotPanel.Alignment="MiddleRight"
                  Fill="LightGreen" go:NodePanel.Figure="TriangleRight"
                  Width="10" Height="10" />
    <go:NodeShape go:SpotPanel.Spot="MiddleBottom"
                  go:SpotPanel.Alignment="MiddleBottom"
                  Fill="LightGreen" go:NodePanel.Figure="TriangleDown"
                  Width="10" Height="10" />
  </go:SpotPanel>
</DataTemplate>
```

Here's what you might see with a node, both unselected and selected using this adornment template:



Selection Appearance Changes

However one can also modify the appearance of a selected part. Basically you can bind properties of your node to values that depend on the **Part.IsSelected** property. For example, you could define a converter that returned a red brush if the input value is true or that returned a normal brush if the value is false.

```
  <go:BooleanBrushConverter x:Key="theSelectedBrushConverter"
                            TrueColor="Red">
    <go:BooleanBrushConverter.FalseBrush>
      <LinearGradientBrush StartPoint="0.5,0" EndPoint="0.5,1">
        <GradientStop Color="White" Offset="0.0" />
```

```
        <GradientStop Color="LightBlue" Offset="1.0" />
      </LinearGradientBrush>
    </go:BooleanBrushConverter.FalseBrush>
  </go:BooleanBrushConverter>
```

In this case, the normal brush is a linear gradient. Now we can bind the **Background** of a panel to the brush returned by this converter based on the value of the part's **IsSelected** property:

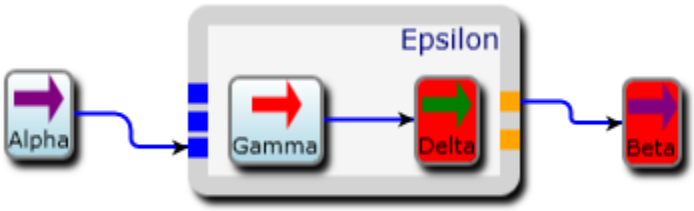
```
<DataTemplate x:Key="NodeTemplate2">
  <!-- note that the binding path is Path=Node.xxx not Path=Data.xxx -->
  <Grid Background="{Binding Path=Node.IsSelected,
                        Converter={StaticResource theSelectedBrushConverter}}">
    . . .
  </Grid>
</DataTemplate>
```

Note how the binding goes to the **Part.IsSelected** property, not to **Data.IsSelected**, because there is no **IsSelected** property on the data class.

As a concrete example:

```
<DataTemplate x:Key="NodeTemplate3">
  <Border BorderBrush="Gray" BorderThickness="2" CornerRadius="5"
          Background="{Binding Path=Part.IsSelected,
                          Converter={StaticResource theSelectedBrushConverter}}"
          go:Node.Location="{Binding Path=Data.Location, Mode=TwoWay}">
    <Border.Effect>
      <DropShadowEffect />
    </Border.Effect>
    <StackPanel Orientation="Vertical">
      <go:NodePanel HorizontalAlignment="Center">
        <Path go:NodePanel.Figure="Arrow" Width="25" Height="25"
              Fill="{Binding Path=Data.Color,
                          Converter={StaticResource theStringBrushConverter}}" />
      </go:NodePanel>
      <TextBlock x:Name="Text" Text="{Binding Path=Data.Key}"
                 HorizontalAlignment="Center" />
    </StackPanel>
  </Border>
</DataTemplate>
```

So when you select "Delta" and "Beta", they appear as follows:



If you want to execute your own code when the selection changes, you can handle the **Diagram.SelectionChanged** event.

Content Alignment and Stretch

The **DiagramPanel** is the panel that holds all of the **Layer**s that together hold all of the **Node**s and **Link**s. The **DiagramPanel** is what supports scrolling around and zooming into the diagram. You can scroll programmatically by setting **DiagramPanel.Position** and you can zoom in or out programmatically by setting **DiagramPanel.Scale**. The user can scroll using the scrollbars or the **PanningTool**, and the user can zoom in or out using Control-Mouse-Wheel.

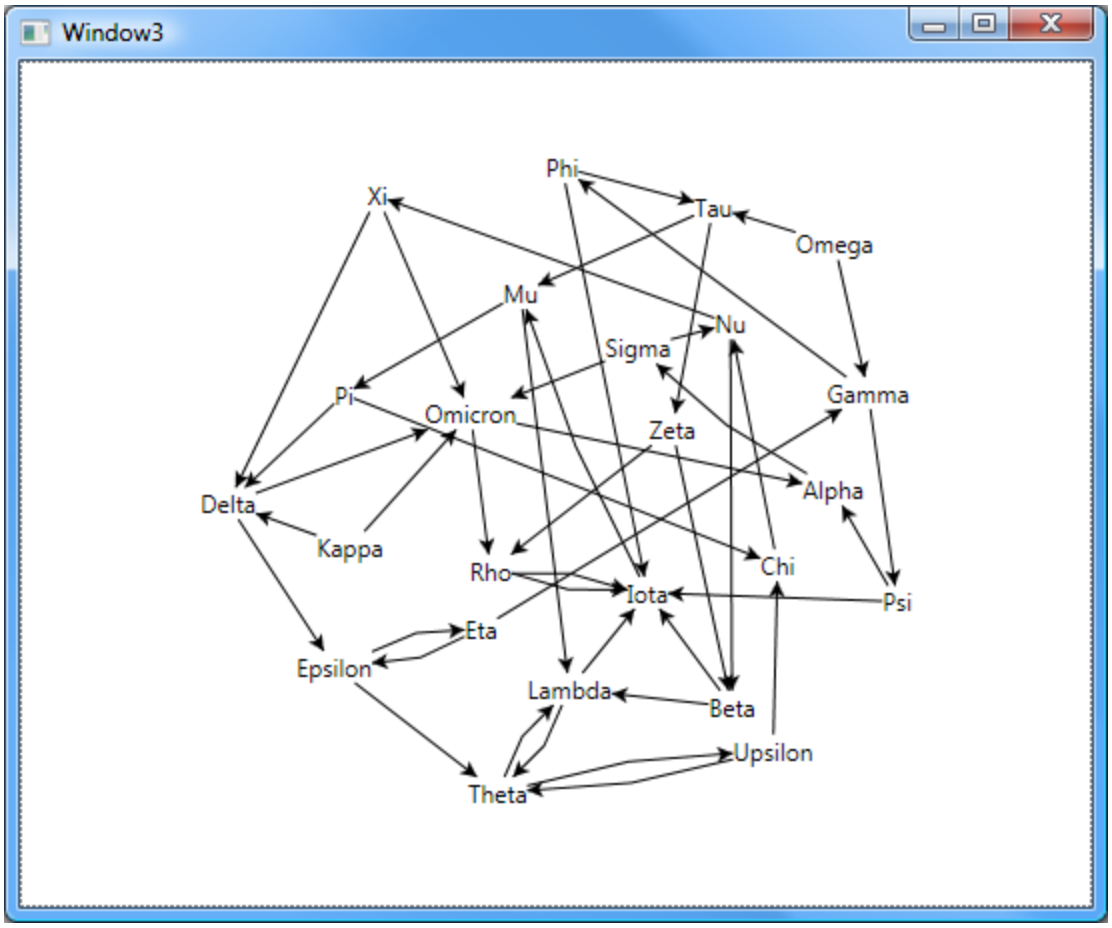
The **DiagramPanel.DiagramBounds** property indicates the total extent of all of the nodes and links. This value is automatically updated as nodes are added or removed. If you do not want the **DiagramBounds** to always reflect the sizes and locations of all of the nodes and links, you can set the **FixedBounds** property. However, if there are any nodes that are located beyond the **FixedBounds**, it is possible that one cannot scroll the diagram to see them.

The **DiagramPanel** has four properties that you will find useful in controlling what is seen and where.

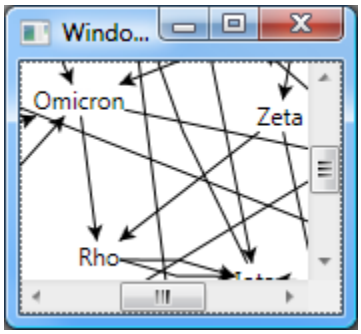
The **HorizontalContentAlignment** and **VerticalContentAlignment** properties determine how the diagram is aligned in the viewport shown by the **DiagramPanel**, when the **DiagramBounds** at the current **Scale** can fit in the viewport. If you want to keep everything centered in the diagram, set both of these properties to “Center”. With the standard **ControlTemplate** you can set these properties on the **Diagram**:

```
<go:Diagram x:Name="myDiagram"
    HorizontalContentAlignment="Center" VerticalContentAlignment="Center" />
```

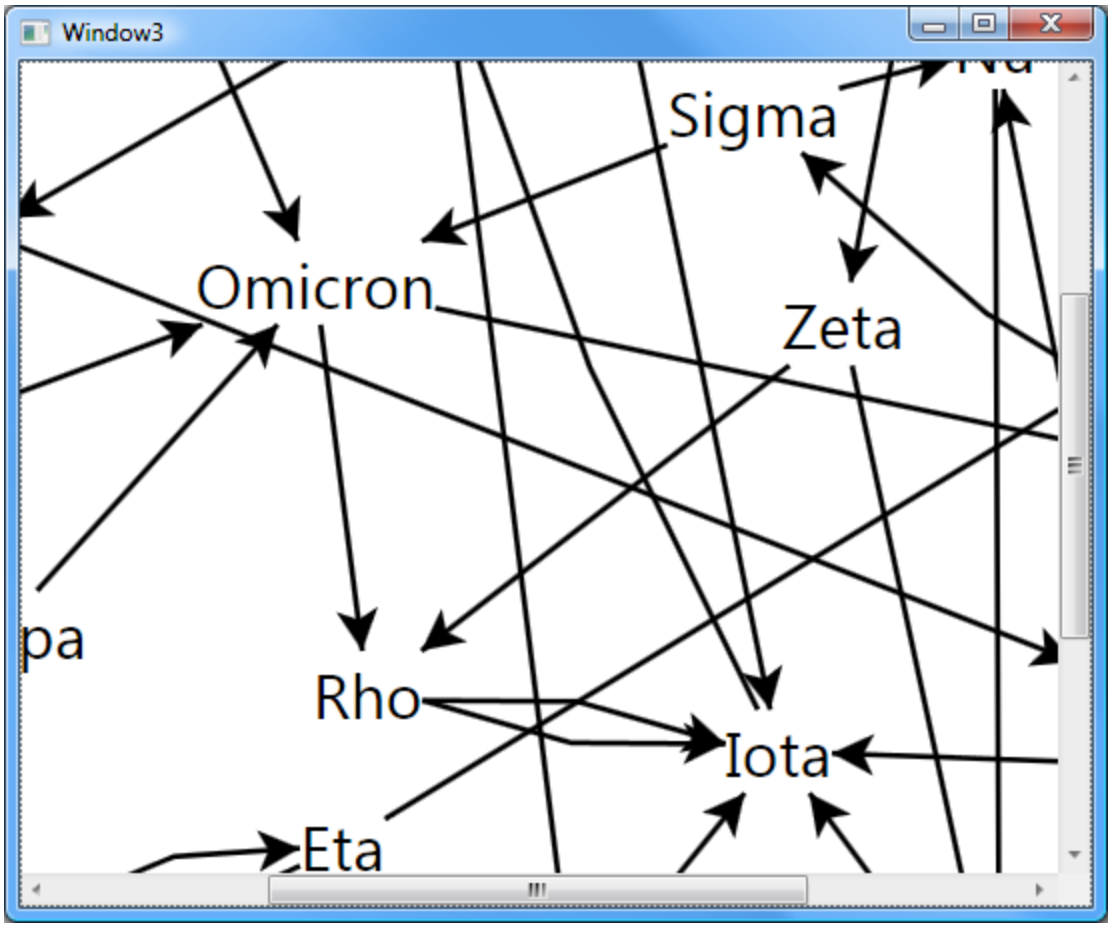
Here’s some random content that fits in the **DiagramPanel** at the current scale**:**



Resize the **Diagram** to be much smaller, and it automatically keeps the center of the diagram centered in the **DiagramPanel** and shows the scrollbars.



Or, leave the **Diagram** size the same, but zoom in with Control-plus or Control-wheel, and it also keeps the center point and shows the scrollbars.

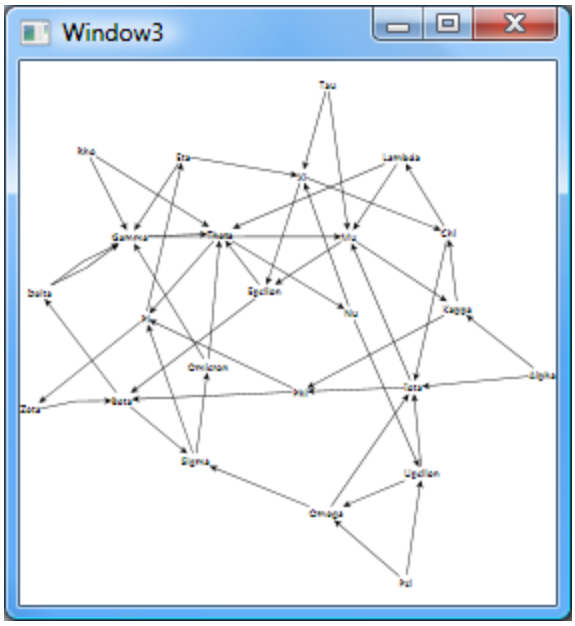


If you don't want the diagram contents to be aligned continuously, use values of **HorizontalAlignment.Stretch** and/or **VerticalAlignment.Stretch**. In this context the meaning of those enumeration values is somewhat different than normal, because the diagram never "stretches" the content. It is common for **Diagram**s to use values of **…Alignment.Stretch** where the user is manually constructing the graph by drag-and-drop.

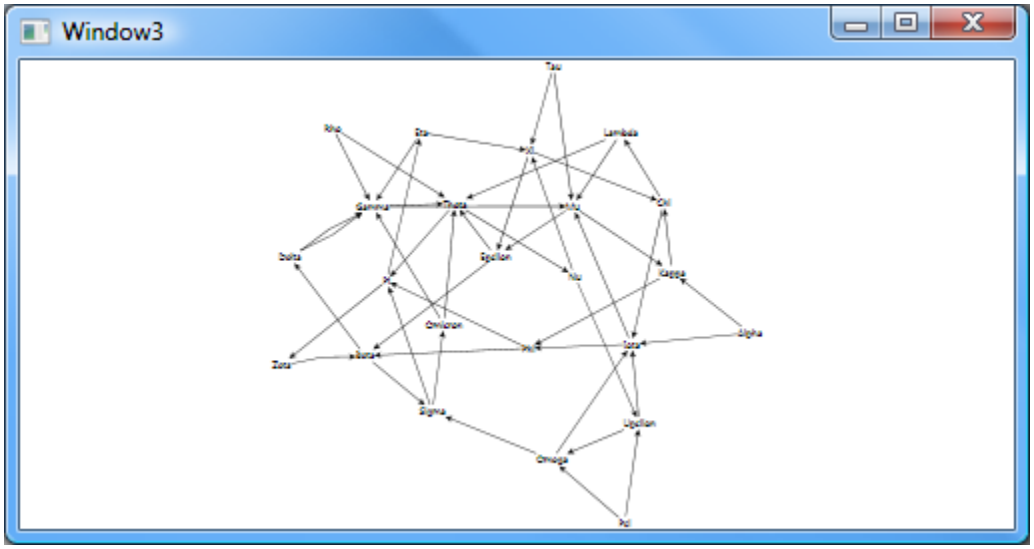
If you want the scale to change automatically as the **Diagram** is resized, use the **DiagramPanel.Stretch** property. (This is not an alignment property, but a property to control the scale of the diagram contents.)

```
<go:Diagram x:Name="myDiagram" Stretch="Uniform"
    HorizontalContentAlignment="Center" VerticalContentAlignment="Center" />
```

This will automatically rescale the diagram so that the whole diagram's bounds fits. You can also use the value of **StretchPolicy.UniformToFill**, which rescales the diagram so that the narrower or the shorter distance fills up the whole area, using a scrollbar to scroll the other dimension (taller or wider). The default value is **StretchPolicy.Unstretched**, which does not change the **DiagramPanel.Scale**.



When there is extra space left over, the contents are centered, according to the two **…Alignment** properties.



Finally, the **DiagramPanel.Padding** property adds a little space to the **DiagramPanel.DiagramBounds**, to avoid having the edge of the **DiagramPanel** come too close to the contents. Because the default value of **Control.Padding** is a **Thickness** of zero on all four sides, we recommend a larger value so that the edges of **Node**s will not appear to bump against the edges of the **Diagram**.

As the default **ControlTemplate** above shows, the four **DiagramPanel** properties (**HorizontalContentAlignment**, **VerticalContentAlignment**, **Stretch**, and **Padding**) normally get their values from the **Diagram**, via **TemplateBinding**s.

If you want to set some properties on a **DiagramPanel** or call its methods, be sure to do so only after the **Diagram.Template** has been applied (i.e. expanded and copied). Until the **ControlTemplate** has been applied, the value of **Diagram.Panel** will be null.

For example if you want to establish an event handler on a **Diagram**'s **Panel**, you can do so in a **Diagram.TemplateApplied** event handler:

```
// wait until the Diagram's Panel exists before establishing its event handler
myDiagram.TemplateApplied += (s, e) => {
  myDiagram.Panel.ViewportBoundsChanged += Panel_ViewportBoundsChanged;
};
```

Initial Positioning and Scaling

The aforementioned **DiagramPanel** properties control the scale (**Stretch)** and position (**HorizontalContentAlignment** and **VerticalContentAlignment**) all the time. However, it is common to want to set the scale and/or position of the diagram after the first layout has positioned all of the nodes, but not thereafter. Towards that end you can set the **Diagram.InitialScale** and/or **Diagram.InitialPosition** properties.

```
<go:Diagram InitialPosition="0 0" . . . >
```

But there are additional **Diagram** properties that are convenient for setting the initial scale and/or position of the **DiagramPanel**. You can set the **Diagram.InitialStretch** property to perform a one-time rescaling. For example:

```
<go:Diagram x:Name="myDiagram" InitialStretch="Uniform"
    HorizontalContentAlignment="Stretch" VerticalContentAlignment="Stretch" />
```

This will perform an initial layout of the contents of the diagram, compute the new **DiagramPanel.DiagramBounds**, and rescale it and position it so that everything fits. Afterwards, the user is free to zoom in or out and to scroll around, as needed.

Two related **Diagram** properties help position the diagram in the panel based on the diagram's bounds: **InitialDiagramBoundsSpot** and **InitialPanelSpot**. The former property specifies which spot of the diagram contents should be positioned, and the latter property specifies where in the **DiagramPanel** it should be positioned. For example:

```
<go:Diagram x:Name="myDiagram"
    InitialDiagramBoundsSpot="MiddleTop" InitialPanelSpot="MiddleTop"
    HorizontalContentAlignment="Stretch" VerticalContentAlignment="Stretch" />
```

This will position the middle-top point of the laid-out diagram at the middle-top point of the panel. You will need to be careful not to choose combinations of values that result in nothing being visible.

DiagramPanel implements the **IScrollInfo** interface, so you can use those methods and properties to scroll programmatically. The **DiagramPanel.MakeVisible** method is useful to scroll the view if the given **Part** is not somewhere in the viewport. The **DiagramPanel.CenterPart** method is useful to try to center a given **Part** in the viewport, although the panel might not be able to scroll that far, especially if the content alignment properties are not "Stretch".

Tools

For flexibility and simplicity, all mouse input is redirected by the **Diagram** to go the diagram's **CurrentTool**. By default the **CurrentTool** is an instance of **ToolManager**, which is responsible for finding another tool that is ready to run and then making it the new **CurrentTool**. This causes the new tool to process mouse events and keyboard events until the tool decides it is finished, at which time the diagram's current tool reverts to the default **ToolManager** tool.

There are a number of predefined tools that each **Diagram** has – they are accessible as diagram properties and can be replaced by setting those properties. The name of the tool class is the same as the name of the diagram property.

Some tools want to run when a mouse-down occurs. These tools include:

- **RelinkingTool**, for reconnecting an existing **Link**
- **LinkReshapingTool**, for changing the route of a **Link**
- **ResizingTool**, for resizing a **Node** or an element within a **Node**
- **RotatingTool**, for rotating a **Node** or an element within a **Node**

Some tools want to run when a mouse-move occurs, after a mouse-down. These tools include:

- **LinkingTool**, for drawing a new **Link**
- **DraggingTool**, for moving or copying selected **Part**s
- **DragSelectingTool**, for rubber-band selection of some **Part**s within a rectangular area
- **PanningTool**, for panning/scrolling the diagram

Some tools only want to run upon a mouse-up event, after a mouse-down. These tools include:

- **TextEditingTool**, for in-place editing of **TextBlock**s in selected **Part**s
- **ClickCreatingTool**, for inserting a new **Node** where the user clicked
- **ClickSelectingTool**, for selecting or de-selecting a **Part**

Finally, there are some tools, such as **DragZoomingTool**, that are not normally invoked by the mouse, but can be started explicitly by setting **Diagram.CurrentTool**.

To change the behavior of a tool, you can set its properties in XAML and replace the corresponding **Diagram** property. For example, to cause control-drag copies to copy the whole effective selection instead of only the selected parts:

```
<go:Diagram . . . >
  <go:Diagram.DraggingTool>
    <gotool:DraggingTool CopiesEffectiveCollection="True" />
  </go:Diagram.DraggingTool>
</go:Diagram>
```

To remove a tool, set it to null. For example, to remove the background rubber-band selection tool:

```
<go:Diagram DragSelectingTool="{x:Null}" . . . />
```

Removing this tool also allows the **PanningTool** to be able to run, because by default the **DragSelectingTool** takes precedence.

As another example, it turns out that the **ClickCreatingTool** is normally never eligible to run because it does not have a value for **ClickCreatingTool.PrototypeData**. You might find it suitable to enable it by setting that property:

```
<go:Diagram . . . >
  <go:Diagram.ClickCreatingTool>
    <go:ClickCreatingTool>
      <go:ClickCreatingTool.PrototypeData>
        <local:MyData Key="Lambda" Color="Fuchsia" />
      </go:ClickCreatingTool.PrototypeData>
    </go:ClickCreatingTool>
  </go:Diagram.ClickCreatingTool>
</go:Diagram>
```

Caution: do not define a tool in XAML as the value of a **Style Setter**, because only one instance of each tool is ever created, and would thus be shared by all diagrams affected by that style. A **DiagramTool** must not be shared by different **Diagram**s.

Events

All of the predefined tools that modify the model do so within a model transaction, and they also raise an event.

Tool	Event
ClickCreatingTool	**NodeCreatedEvent**
DraggingTool	**SelectionMovedEvent** or **SelectionCopiedEvent** or **ExternalObjectsDroppedEvent**
LinkingTool	**LinkDrawnEvent**
RelinkingTool	**LinkRelinkedEvent**

LinkReshapingTool	**LinkReshapedEvent**
ResizingTool	**NodeResizedEvent**
RotatingTool	**NodeRotatedEvent**
TextEditingTool	**TextEditedEvent**

The other predefined tools do not have model-changing side-effects.

There are a number of events raised by commands, implemented by the **CommandHandler**.

Command	Event
Delete **Cut**	**SelectionDeletingEvent** and **SelectionDeletedEvent**
Paste	**ClipboardPastedEvent**
Group	**SelectionGroupedEvent**
Ungroup	**SelectionUngroupedEvent**

The other predefined commands do not have model-changing side-effects.

All of these events are defined on the **Diagram** class. Events are implemented as **RoutedEvent**s.

Mouse Clicks

A simple mouse click on a selectable **Part** will result in that part becoming selected. This is the behavior of the **ClickSelectingTool**.

Remember also that if you want to update some displays based on the currently selected node, you can data-bind the **Diagram.SelectedNode** property. This was discussed in the section about selection.

If you want to perform some custom action when the user double-clicks on a part, you can define an event handler for the part:

```
<DataTemplate x:Key="NodeTemplate4">
  <Border . . .
          MouseLeftButtonDown="Node_MouseLeftButtonDown">
    . . .
  </Border>
</DataTemplate>

private void Node_MouseLeftButtonDown(object sender, MouseButtonEventArgs e) {
  if (DiagramPanel.IsDoubleClick(e)) {
    Node node = Part.FindAncestor<Node>(sender as UIElement);
    if (node != null && node.Data != null) {
      e.Handled = true;
      MessageBox.Show("double clicked on " + node.Data.ToString());
    }
  }
}
```

You can implement context menus for nodes by just defining them in your node template:

```
<DataTemplate x:Key="NodeTemplate3">
```

```
  <Border . . .>
    <ContextMenuService.ContextMenu>
      <ContextMenu>
        <MenuItem Header="some node command" Click="MenuItem_Click" />
      </ContextMenu>
    </ContextMenuService.ContextMenu>
  </Border>
</DataTemplate>

private void MenuItem_Click(object sender, RoutedEventArgs e) {
  var partdata = ((FrameworkElement)sender).DataContext as PartManager.PartBinding;
  if (partdata == null || partdata.Data == null) {
    MessageBox.Show("Clicked on nothing or on unbound part");
  } else {
    MessageBox.Show("Clicked on data: " + partdata.Data.ToString());
  }
}
```

Other Events

The **Diagram** and **DiagramPanel** classes provide a number of additional events not necessarily related to tools or commands.

The **Diagram.InitialLayoutCompleted** event is raised after the **LayoutManager** has performed the first layout(s) after the **Diagram**'s template has been applied and the **DiagramPanel.DiagramBounds** has been updated. It can happen again when the **Diagram.Model** is replaced.

The **Diagram.LayoutCompleted** event is raised when the **LayoutManager** has finished performing all needed diagram and group layouts and the **DiagramPanel.DiagramBounds** property has been updated. The frequency of this event depends on the how often layouts need to be performed: the value of **DiagramLayout.Conditions**, when nodes or links being added or removed or resized, and explicit calls to **Diagram.LayoutDiagram()**. This event occurs after any **InitialLayoutCompleted** event.

> *Note: you should ignore the* **UIElement.LayoutUpdated** *event. It has nothing to do with diagram layout.*

The **Diagram.ModelReplaced** event is raised when the **Diagram.Model** value is replaced.

The **Diagram.SelectionChanged** event is raised when the contents of the **Diagram.SelectedParts** collection changes. You might not need to implement such an event handler if you can depend on data-bindings of the **Diagram.SelectedNode**, **SelectedLink**, and/or **SelectedGroup** dependency properties.

The **Diagram.TemplateApplied** event is raised after the **Diagram**'s **ControlTemplate** has been expanded. This is convenient for initializing the **Diagram.Panel**, which will not exist until the template is applied.

The **Diagram.TemplatesChanged** event is raised whenever any of the **DataTemplate**s of the **Diagram** is replaced, including the template dictionary properties. However, modifying the contents of a **DataTemplateDictionary** will not raise any events.

The **DiagramPanel.DiagramBoundsChanged** event is raised when the value of **DiagramPanel.DiagramBounds** has changed.

The **DiagramPanel.ViewportBoundsChanged** event is raised when the value of **DiagramPanel.ViewportBounds** has changed. That happens when the **DiagramPanel.Position** or **Scale** or **Width** or **Height** properties have changed.

The **Control.Unloaded** event by default causes the **PartManager** to discard all of the **Node**s and **Link**s. The **Control.Loaded** event rebuilds all of the **Node**s and **Link**s. This may result is some loss of state or other side-effects when the **Diagram** is removed from the visual tree and then re-inserted into it. Typically this will happen when the **Diagram** is in a tab of a **TabControl** and the user switches tabs. We suggest that you set **Diagram.UnloadingClearsPartManager** to false in order to avoid the clearing of **Part**s and their rebuilding.

Commands

The **Diagram** control also supports various commands. The **CommandHandler** class implements pairs of methods: a method to execute a command, and a predicate that is true when the command may be executed. For example, for the **Copy** command, there is a **CommandHandler.Copy()** method and a **CommandHandler.CanCopy()** method. These are virtual methods so that you can easily customize their behavior by overriding them and replacing the value of **Diagram.CommandHandler**.

WPF offers support for routed commands.

Non-routed **ICommand**: property of **CommandHandler** class	**RoutedCommand**: static property of **Commands** class
CopyCommand	**Copy**
CutCommand	**Cut**
DeleteCommand	**Delete**
PasteCommand	**Paste**
PrintCommand	**Print**
RedoCommand	**Redo**
SelectAllCommand	**SelectAll**
UndoCommand	**Undo**
DecreaseZoomCommand	**DecreaseZoom**

IncreaseZoomCommand	**IncreaseZoom**
ZoomCommand	**Zoom**
GroupCommand	**Group**
UngroupCommand	**Ungroup**
EditCommand	**Edit**

Using non-routed commands:

```
<Button Command="{Binding ElementName=myDiagram,
                          Path=CommandHandler.CopyCommand}">Copy</Button>
```

Using routed commands:

```
<Button Command="Copy"
        CommandTarget="{Binding ElementName=myDiagram}">Copy</Button>
```

User Permissions

Programmatically there is no restriction on the kind of operation that you may perform. However, you may want to restrict the actions that your users may perform.

The simplest restriction is to set the **Diagram** property **IsEnabled** to false. Users will not be able to do much of anything.

More common is to set **Diagram.IsReadOnly** to true. This allows users to scroll and zoom and to select parts, but not to insert or delete or drag or modify parts. Caution: just because the diagram is read-only doesn't mean all controls in your templates are read-only – most controls do not even have that concept.

More precise restrictions can be imposed by setting to false properties of the **Diagram** or of a particular **Layer**. Some restrictions, such as **AllowZoom**, only make sense when applying to the whole **Diagram**. Others may also apply to individual parts, such as **Copyable** on **Part** corresponding to **Diagram.AllowCopy** and to **Layer.AllowCopy**.

Most of these boolean properties are true by default. Exceptions include: **Diagram.AllowDragOut**, **Diagram.AllowDrop**, and **Part.Reshapable/Resizable/Rotatable**, **Part.TextEditable**, **Node.Linkable…**, and **Link.Relinkable…**, which are all false by default.

Enabled Action	**Diagram** Property	**Layer** property	**Part** (**Node**, **Group**, or **Link**) attached properties on root visual element of data template
Cut/Copy/Paste commands	**AllowClipboard**		
Cut/Copy commands, control-drag copy	**AllowCopy**	**AllowCopy**	**Copyable**
Cut/Delete/Ungroup commands	**AllowDelete**	**AllowDelete**	**Deletable**

Drag-and-drop out of diagram	**AllowDragOut**		
Drag-and-drop into diagram	**AllowDrop**		
TextEditingTool	**AllowEdit**	**AllowEdit**	**Editable** (on a Part) and **TextEditable** (on a TextBlock)
Group command	**AllowGroup**	**AllowGroup**	**Groupable**
Group/Paste commands, ClickCreatingTool, DraggingTool	**AllowInsert**		
LinkingTool	**AllowLink**	**AllowLink**	**Node.LinkableFrom**, **Node.LinkableTo**, **Node.LinkableSelfNode**, **Node.LinkableDuplicates**, **Node.LinkableMaximum** (all on any port element)
DraggingTool	**AllowMove, AllowCopy, AllowInsert**	**AllowMove, AllowCopy**	**Movable, Copyable, DragOverSnapEnabled, DragOverSnapCellSize, DragOverSnapCellSpot, Node.MinLocation**, **Node.MaxLocation**
Printing	**AllowPrint**	**AllowPrint**	**Printable**
RelinkingTool	**AllowRelink**	**AllowRelink**	**Route.RelinkableFrom**, **Route.RelinkableTo**
LinkReshapingTool	**AllowReshape**	**AllowReshape**	**Reshapable**
ResizingTool	**AllowResize**	**AllowResize**	**Resizable**
RotatingTool	**AllowRotate**	**AllowRotate**	**Rotatable**
PanningTool, DiagramPanel scrolling	**AllowScroll**		
SelectAll command, ClickSelectingTool, DragSelectingTool	**AllowSelect**	**AllowSelect**	**Selectable**
Undo/Redo commands	**AllowUndo**		
Ungroup command	**AllowUngroup**	**AllowUngroup**	**Group.Ungroupable**
Zoom commands	**AllowZoom**		

In addition to the diagram/layer/part properties listed above, there are some relevant properties on some of the model classes.

The **DiagramModel.Modifiable** property controls whether the user may create or delete or modify nodes or links or groups. ***Caution: this property is false by default!*** Note also that a value of false will only disable changes to the model, not necessarily to your data. The model cannot know about, nor can it affect, all data binding or programmatic changes that you do directly to the data. For example, a false value for **DiagramModel.Modifiable** will not prevent users from moving nodes around, because the model does not have any knowledge about node positions.

The **GraphLinksModel.ValidCycle** and **GraphModel.ValidCycle** properties control what kinds of graphs the model supports. This in turn may limit which links may be drawn or reconnected by the user. By default there are no restrictions on creating cycles in graphs.

The **DiagramModel.HasUndoManager** property is false by default. You may set this to true in order to enable undo and redo of model changes and data property changes. You probably will want to set this after initializing the model, so that users cannot undo your model setup.

Please note that because the **UndoManager** only records model and data property changes, if you want the user to be able to undo/redo the drag-moving of nodes, you will need to bind the **Node.Location** attached property in your node and group templates to your node data property. For example:

```
go:Node.Location="{Binding Path=Data.Location, Mode=TwoWay}"
```

Link Validation

In many diagrams there are semantic restrictions on which links could be considered "valid".

A good user interface will try to prevent the user from drawing invalid links. This is much friendlier than permitting "bad" links and then trying to point out the errors much later.

There are two linking tools: **LinkingTool** for drawing new links and **RelinkingTool** for reconnecting existing links. There are several built-in properties and methods that will help you constrain the links that the user may create using these tools.

No links can be drawn by the user unless the **Diagram** has a **LinkingTool** and there are nodes with valid ports from which the user can draw new links. So you can easily prevent new links from being drawn by:

- not having any elements in your node **DataTemplate** acting as valid ports, **or**
- not setting **DiagramModel.Modifiable** to true, **or**
- making the **Diagram.IsReadOnly** or setting **Diagram.AllowLink** false, **or**
- removing the **LinkingTool** by setting **Diagram.LinkingTool** to null in either XAML or code

The first condition holds true by default. If you want users to draw new links interactively, say from node A to node B, you have to make sure that node A has at least one **FrameworkElement** with the **Node.LinkableFrom** attached property set to true, and that node B has at least one **FrameworkElement** with the **Node.LinkableTo** attached property set to true. (Remember that these **Linkable...** attached properties should be set on either the root **FrameworkElement** of the **DataTemplate** or on any **FrameworkElement** that is declared to be a “port” by setting the **Node.PortId** attached property, as discussed in the section about ports on nodes.)

No links can be reconnected by users unless the **Diagram** has a **RelinkingTool** and there are links with a "Relinkable..." property set to true and there are valid nodes to connect to. You can prevent users from reconnecting existing links by:

- not setting the **Route.RelinkableFrom** and **Route.RelinkableTo** properties in your link **DataTemplate** to true, **or**
- not setting **DiagramModel.Modifiable** to true, **or**
- making the **Diagram.IsReadOnly** or setting **Diagram.AllowRelink** false, **or**
- removing the **RelinkingTool** by setting **Diagram.RelinkingTool** to null in either XAML or code

But often one desires constraints on the permitted links in a diagram. There are some predefined properties that are convenient for declaring certain cases, and there are some methods that you can override for the general situation.

The **Node.LinkableSelfNode** attached property can be set to true on **FrameworkElement**s acting as ports in order to permit both ends of a link to be the same node. (By default reflexive links are not allowed.)

The **Node.LinkableDuplicates** attached property can be set to true on **FrameworkElement**s acting as ports in order to allow more than one link connecting the same two ports in the same direction. (By default multiple links are not allowed.)

The **Node.LinkableMaximum** attached property can be set on **FrameworkElement**s acting as ports to limit how many links can be connected to that port in either direction. (By default there is no limit.)

There is also the **ValidCycle** property on **GraphModel** and **GraphLinksModel** that describes what kinds of graphs are allowed. (By default all kinds of graphs are permitted.)

Finally, for the most general case, where there are application-specific reasons for allowing some links and disallowing others, you can override the **IsValidLink** method in both linking tools.

For example, say that you want to change the behavior of the Flow Chart sample to disallow the user from specifying links that go directly from "Start" nodes to "End" nodes. You can achieve this by adding the following override of **IsValidLink** to both of the custom linking tools that the Flow Chart sample defines:

```
public override bool IsValidLink(Node fromnode, FrameworkElement fromport,
                                 Node tonode, FrameworkElement toport) {
  if (!base.IsValidLink(fromnode, fromport, tonode, toport)) return false;
  // don't allow a link directly from Start to End
  MyNodeData fromnodedata = fromnode.Data as MyNodeData;
```

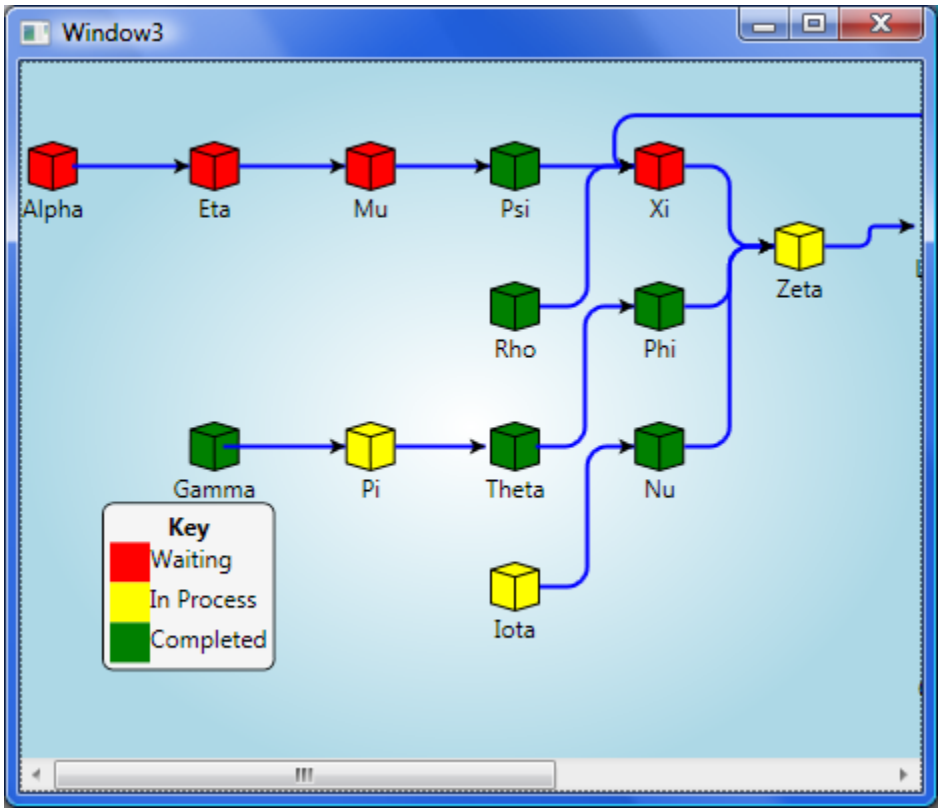
```
  MyNodeData tonodedata = tonode.Data as MyNodeData;
  if (fromnodedata != null && fromnodedata.Category == "Start" &&
      tonodedata != null && tonodedata.Category == "End") return false;
  return true;
}
```

Of course this is just a simple example. You can implement arbitrarily complex predicates that examine your application data.

Diagram Background and Grids

Since a **Diagram** is a regular **Control**, you can set its **Background** property as you can with any other control:

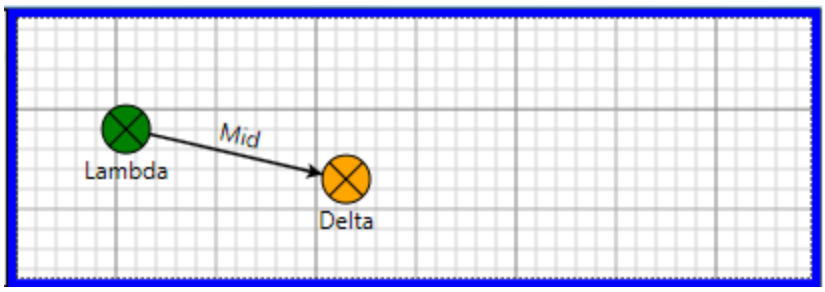
```
<go:Diagram . . .>
  <Control.Background>
    <RadialGradientBrush>
      <GradientStop Color="White" Offset="0.0" />
      <GradientStop Color="LightBlue" Offset="1.0" />
    </RadialGradientBrush>
  </Control.Background>
</go:Diagram>
```



Grids

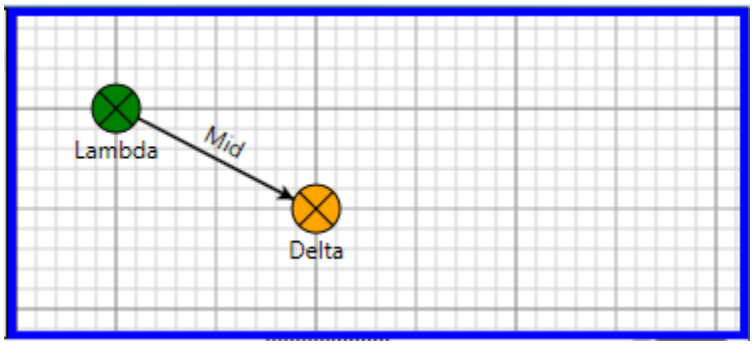
Just by setting **Diagram.GridVisible** to true, you will show a default grid for the whole diagram.

```
<go:Diagram GridVisible="True" BorderBrush="Blue" BorderThickness="4" ... />
```



If you also set the various **Diagram.GridSnap…** properties, you will affect how the **DraggingTool** operates.

```
<go:Diagram GridVisible="True" BorderBrush="Blue" BorderThickness="4"
            GridSnapEnabled="True" GridSnapCellSize="50 50" ... />
```



Grid snapping involves setting the **Node.Location** to grid points. In this example the **Diagram.NodeTemplate** has redefined the "Location" to be the center of the colored circle, by setting the **Node.LocationElementName** and **Node.LocationSpot** properties:

```
<DataTemplate x:Key="NodeTemplate">
  <StackPanel go:Part.Text="{Binding Path=Data.Key}"
              go:Node.Location="{Binding Path=Data.Location, Mode=TwoWay}"
              go:Node.LocationElementName="Icon" go:Node.LocationSpot="Center"
              go:Part.SelectionElementName="Icon"
              go:Part.Resizable="True" go:Part.ResizeCellSize="10 10">
    <go:NodeShape x:Name="Icon" go:NodePanel.Figure="Junction"
            HorizontalAlignment="Center" Width="25" Height="25"
            Stroke="Black" StrokeThickness="1"
            Fill="{Binding Path=Data.Color,
                    Converter={StaticResource theStringBrushConverter}}"
            go:Node.PortId=""
            go:Node.LinkableFrom="True" go:Node.LinkableTo="True"
            go:Node.FromSpot="None" go:Node.ToSpot="None" />
    <TextBlock Text="{Binding Path=Data.Key}" HorizontalAlignment="Center" />
  </StackPanel>
</DataTemplate>
```

Hence the centers of the **Ellipse**s are positioned at the grid points that are multiples of 50x50, even though the grid itself happens to show grid lines every 10x10.

This **NodeTemplate** also enables user-resizing of the ellipses of selected nodes, by setting **Part.Resizable** to true and **Part.SelectionElementName**. You can also control the sizes that the **ResizingTool** will permit, by setting **Part.ResizeCellSize**. By default this would be the value of **Diagram.GridSnapCellSize** if **Diagram.GridSnapEnabled** is true.

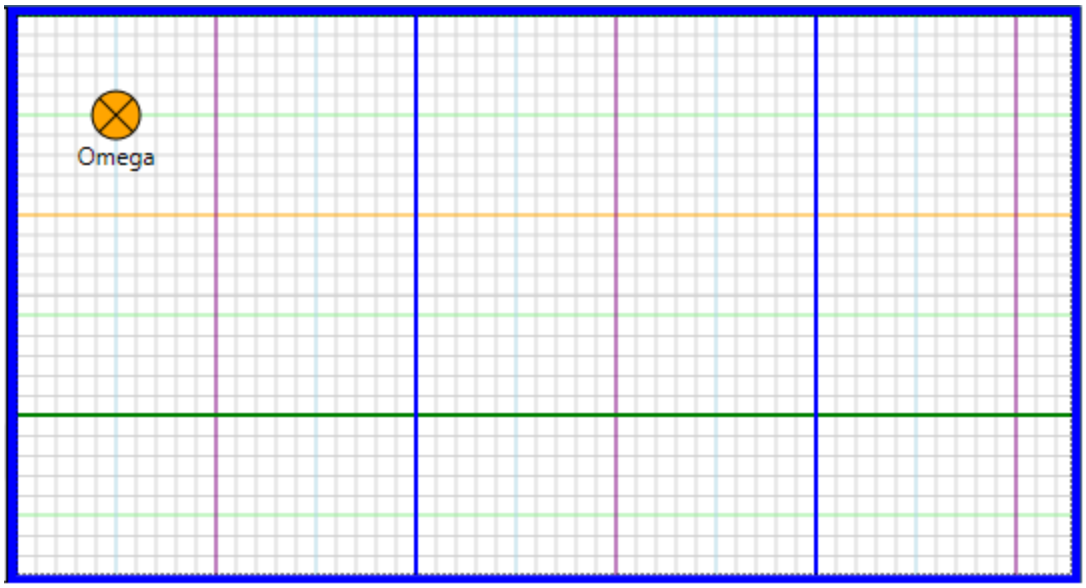
ResizingTool also respects the values of **FrameworkElement.MinWidth**, **MinHeight**, **MaxWidth**, and **MaxHeight.** Those Min/Max… attributes should be set on the element being resized, not on the root visual element for the node.

Custom Grids

Grids are rendered by **GridPattern**s, which allow for great flexibility in producing customized grids. You just need to provide a value for **Diagram.GridPattern**, for example as a property element in XAML:

```
<go:Diagram GridVisible="True" BorderBrush="Blue" BorderThickness="4" ...>
  <go:Diagram.GridPattern>
    <go:GridPattern CellSize="10 10">
      <Path Stroke="LightGray" StrokeThickness="1"
          go:GridPattern.Figure="HorizontalLine" />
      <Path Stroke="LightGray" StrokeThickness="1"
          go:GridPattern.Figure="VerticalLine" />
      <Path Stroke="LightGreen" StrokeThickness="1"
          go:GridPattern.Figure="HorizontalLine" go:GridPattern.Interval="5" />
      <Path Stroke="LightBlue" StrokeThickness="1"
          go:GridPattern.Figure="VerticalLine" go:GridPattern.Interval="5" />
      <Path Stroke="Orange" StrokeThickness="1"
          go:GridPattern.Figure="HorizontalLine" go:GridPattern.Interval="10" />
      <Path Stroke="Purple" StrokeThickness="1"
          go:GridPattern.Figure="VerticalLine" go:GridPattern.Interval="10" />
      <Path Stroke="Green" StrokeThickness="2"
          go:GridPattern.Figure="HorizontalLine" go:GridPattern.Interval="20" />
      <Path Stroke="Blue" StrokeThickness="2"
          go:GridPattern.Figure="VerticalLine" go:GridPattern.Interval="20" />
    </go:GridPattern>
  </go:Diagram.GridPattern>
</go:Diagram>
```

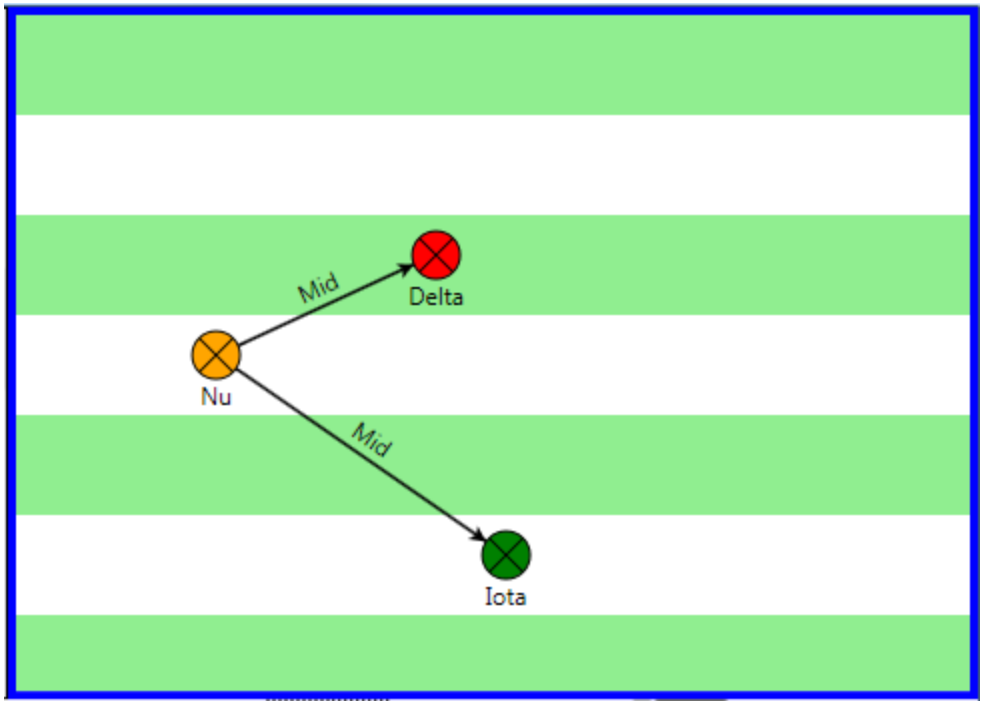
Note how the **GridPattern.Figure** and **GridPattern.Interval** attached properties control the appearance of the grid. The **Interval** property controls how often that **Path** is drawn. All distances between lines are multiples of the **GridPattern.CellSize**.



Instead of supplying a value for **Diagram.GridPattern**, you can set its **Diagram.GridPatternTemplate**, which may be more convenient when wanting to style the **Diagram**. If neither is specified when **Diagram.GridVisible** is true, it uses a default grid pattern template.

For a completely different appearance, you might have only horizontal bars (no vertical lines or bars):

```
<go:Diagram.GridPattern>
  <go:GridPattern CellSize="50 50">
    <Path Fill="LightGreen"
        go:GridPattern.Figure="HorizontalBar" go:GridPattern.Interval="2" />
  </go:GridPattern>
</go:Diagram.GridPattern>
```



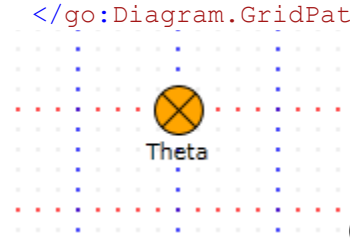
Or show only dots:

```
<go:Diagram.GridPattern>
  <go:GridPattern CellSize="10 10">
    <Path Stroke="Gray" go:GridPattern.Figure="HorizontalDot" />
  </go:GridPattern>
</go:Diagram.GridPattern>
```

(the dots might not be visible in this screenshot)

You can control the size of each dot by setting the **StrokeThickness**:

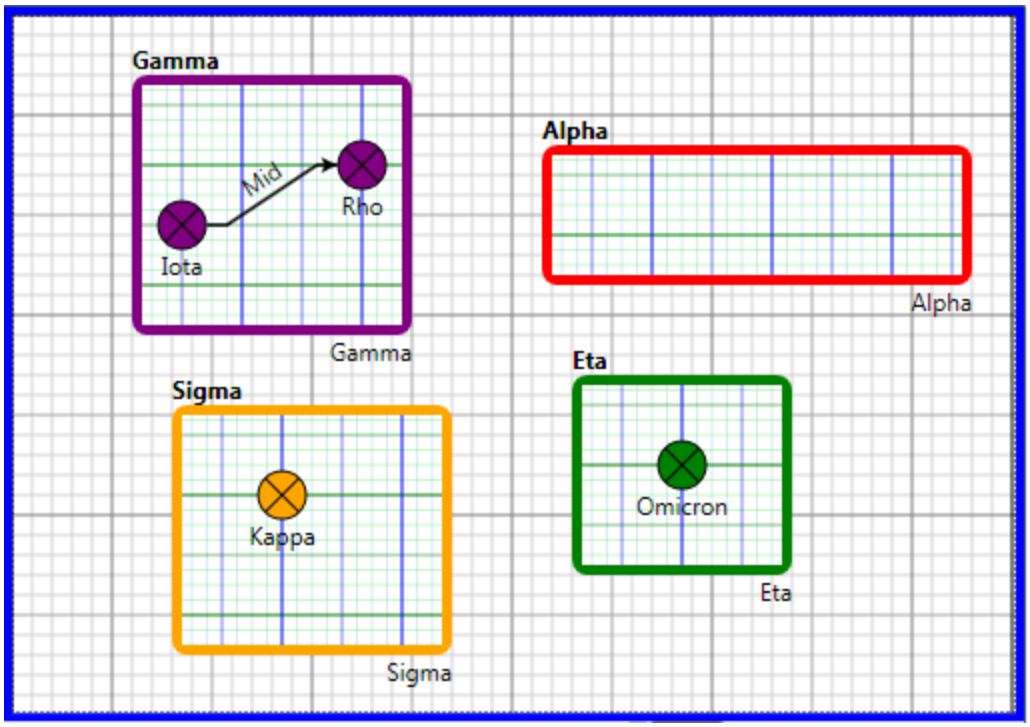
```
<go:Diagram.GridPattern>
  <go:GridPattern CellSize="10 10">
    <Path Stroke="Gray" go:GridPattern.Figure="HorizontalDot" />
    <Path Stroke="Red" go:GridPattern.Figure="HorizontalDot"
          StrokeThickness="2" go:GridPattern.Interval="5" />
    <Path Stroke="Blue" go:GridPattern.Figure="VerticalDot"
          StrokeThickness="2" go:GridPattern.Interval="5" />
  </go:GridPattern>
</go:Diagram.GridPattern>
```



(the smallest dots might not be visible in this screenshot)

Note that **GridPattern** is a **Panel**; you can use it inside any element. For example, the **GroupTemplate** might include a **GridPattern** that is different from the **Diagram.GridPattern**.

```
<DataTemplate x:Key="GroupTemplate">
  <StackPanel go:Node.Location="{Binding Path=Data.Location, Mode=TwoWay}"
              go:Node.LocationElementName="grid"
              go:Part.SelectionElementName="grid" go:Part.Resizable="True"
              go:Part.DragOverSnapEnabled="True"
              go:Part.DragOverSnapCellSize="15 15">
    <TextBlock Text="{Binding Path=Data.Key}" FontWeight="Bold"
               HorizontalAlignment="Left" />
    <Border BorderBrush="{Binding Path=Data.Color,
                    Converter={StaticResource theStringBrushConverter}}"
            BorderThickness="5" CornerRadius="5">
      <go:GridPattern x:Name="grid" CellSize="7.5 7.5"
                      MinWidth="15" MinHeight="15"
                      Width="{Binding Path=Data.Width, Mode=TwoWay}"
                      Height="{Binding Path=Data.Height, Mode=TwoWay}">
        <Path Stroke="LightGreen" StrokeThickness="0.5"
            go:GridPattern.Figure="HorizontalLine" />
        <Path Stroke="LightBlue" StrokeThickness="0.5"
            go:GridPattern.Figure="VerticalLine" />
        <Path Stroke="Green" StrokeThickness="0.5"
            go:GridPattern.Figure="HorizontalLine" go:GridPattern.Interval="4"/>
        <Path Stroke="Blue" StrokeThickness="0.5"
            go:GridPattern.Figure="VerticalLine" go:GridPattern.Interval="4"/>
        <Path Stroke="Green" StrokeThickness="1"
            go:GridPattern.Figure="HorizontalLine" go:GridPattern.Interval="8"/>
        <Path Stroke="Blue" StrokeThickness="1"
            go:GridPattern.Figure="VerticalLine" go:GridPattern.Interval="8"/>
      </go:GridPattern>
    </Border>
    <TextBlock Text="{Binding Path=Data.Key}" HorizontalAlignment="Right" />
  </StackPanel>
</DataTemplate>
```

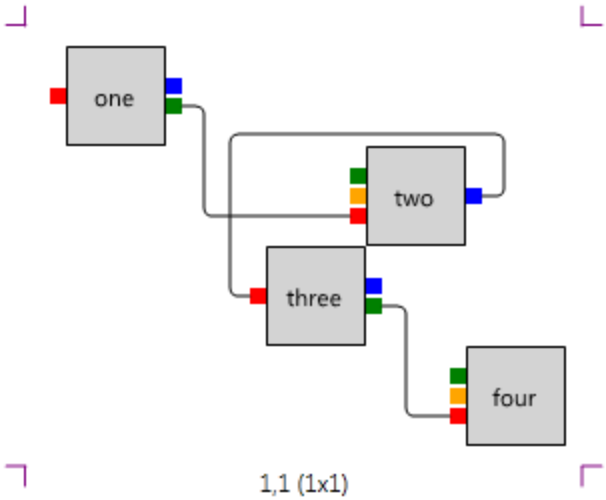


Printing

The **PrintManager** has a number of options for controlling what is printed and how. By default it will print the whole diagram at the standard scale, using as many pages as needed. Or you can set the **PrintManager.Scale** to Double.NaN to have it automatically shrink the scale, if needed, to fit a single page.

```
<go:Diagram . . .>
  <go:Diagram.PrintManager>
    <go:PrintManager Scale="NaN" />
  </go:Diagram.PrintManager>
</go:Diagram>
```

Each page includes an X/Y page number along with the total number of columns and rows of pages. Each page also includes cut marks so that you can easily trim off the right and bottom sides and tape together a large continuous diagram.



You can control what area of the diagram, in model coordinates, is printed by setting **Diagram.PrintManager.Bounds**. But it's more common to let the user select what they want to print, by setting **PrintManager.Parts** in the initialization:

```
myDiagram.PrintManager.Parts = myDiagram.SelectedParts;
```

When nothing is selected, it prints the whole diagram. Note that **Part**s for which **Printable** is false, parts in **Layer**s for which **AllowPrint** is false, and **Adornment**s are not printed.

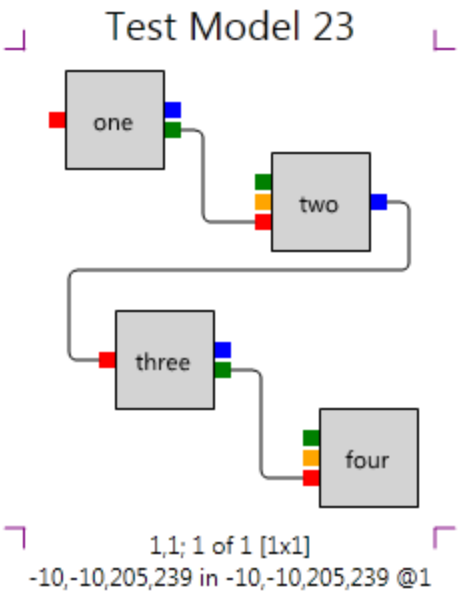
You can also customize the decorations that are printed on each page. By default the **PrintManager.ForegroundTemplate** property is the template that you can see in the Generic.XAML file. But you can define your own. For example:

```
<DataTemplate x:Key="PrintBorderTemplate">
  <go:SpotPanel>  <!-- takes the size of each printed page -->
    <!-- header -->
    <TextBlock Text="{Binding Diagram.Model.Name}" FontSize="20"
          go:SpotPanel.Spot="MiddleTop" go:SpotPanel.Alignment="MiddleBottom" />

    <!-- cut marks -->
    <Line X1="-10" Y1="0" X2="0" Y2="0" Stroke="Purple" StrokeThickness="1"
          go:SpotPanel.Spot="TopLeft" go:SpotPanel.Alignment="BottomRight" />
    <Line X1="0" Y1="-10" X2="0" Y2="0" Stroke="Purple" StrokeThickness="1"
          go:SpotPanel.Spot="TopLeft" go:SpotPanel.Alignment="BottomRight" />
    <Line X1="10" Y1="0"  X2="0" Y2="0" Stroke="Purple" StrokeThickness="1"
          go:SpotPanel.Spot="TopRight" go:SpotPanel.Alignment="BottomLeft" />
    <Line X1="0" Y1="-10" X2="0" Y2="0" Stroke="Purple" StrokeThickness="1"
          go:SpotPanel.Spot="TopRight" go:SpotPanel.Alignment="BottomLeft" />
    <Line X1="10" Y1="0"  X2="0" Y2="0" Stroke="Purple" StrokeThickness="1"
          go:SpotPanel.Spot="BottomRight" go:SpotPanel.Alignment="TopLeft" />
    <Line X1="0" Y1="10"  X2="0" Y2="0" Stroke="Purple" StrokeThickness="1"
          go:SpotPanel.Spot="BottomRight" go:SpotPanel.Alignment="TopLeft" />
    <Line X1="-10" Y1="0" X2="0" Y2="0" Stroke="Purple" StrokeThickness="1"
          go:SpotPanel.Spot="BottomLeft" go:SpotPanel.Alignment="TopRight" />
    <Line X1="0" Y1="10"  X2="0" Y2="0" Stroke="Purple" StrokeThickness="1"
```

```
          go:SpotPanel.Spot="BottomLeft" go:SpotPanel.Alignment="TopRight" />

    <!-- footer -->
    <StackPanel go:SpotPanel.Spot="MiddleBottom"
                go:SpotPanel.Alignment="MiddleTop">
      <StackPanel Orientation="Horizontal" HorizontalAlignment="Center">
        <TextBlock Text="{Binding Column}" />
        <TextBlock Text="," />
        <TextBlock Text="{Binding Row}" />
        <TextBlock Text="; " />
        <TextBlock Text="{Binding Index}" />
        <TextBlock Text=" of " />
        <TextBlock Text="{Binding Count}" />
        <TextBlock Text=" [" />
        <TextBlock Text="{Binding ColumnCount}" />
        <TextBlock Text="x" />
        <TextBlock Text="{Binding RowCount}" />
        <TextBlock Text="]" />
      </StackPanel>
      <StackPanel Orientation="Horizontal" HorizontalAlignment="Center">
        <TextBlock Text="{Binding ViewportBounds}" />
        <TextBlock Text=" in " />
        <TextBlock Text="{Binding TotalBounds}" />
        <TextBlock Text=" @" />
        <TextBlock Text="{Binding Scale}" />
      </StackPanel>
    </StackPanel>
  </go:SpotPanel>
</DataTemplate>
```

Because these print templates are expanded for each page and data-bound to an instance of **PageInfo** that describes that page, you can easily include the page number(s). This example template also shows the **DiagramModel.Name** as the header, which property you will need to set in your model-building code.

```
<go:Diagram x:Name="myDiagram" . . . >
  <go:Diagram.PrintManager>
    <go:PrintManager ForegroundTemplate="{StaticResource PrintBorderTemplate}"
                     Margin="30 70 30 70" />
  </go:Diagram.PrintManager>
</go:Diagram>
```



If you are displaying a background **GridPattern**, you may want the grid to cover all of each printed page. The **PrintManager.PageOptions** property controls whether the **Diagram.Background** and **Diagram.GridPattern** are printed and how much of each page they cover.

```
<go:Diagram x:Name="myDiagram" GridVisible="True" . . . >
  <go:Diagram.PrintManager>
    <go:PrintManager ForegroundTemplate="{StaticResource PrintBorderTemplate}"
                     PageOptions="FullGrid" Margin="30 70 30 70" />
  </go:Diagram.PrintManager>
</go:Diagram>
```

Here is the result of printing a larger diagram with a background grid. This is a screenshot of XPS Viewer at 20% so that all four pages could fit in a reasonable size image for this document. It may be hard to read here, but note the custom header and footer on each page.

Overview

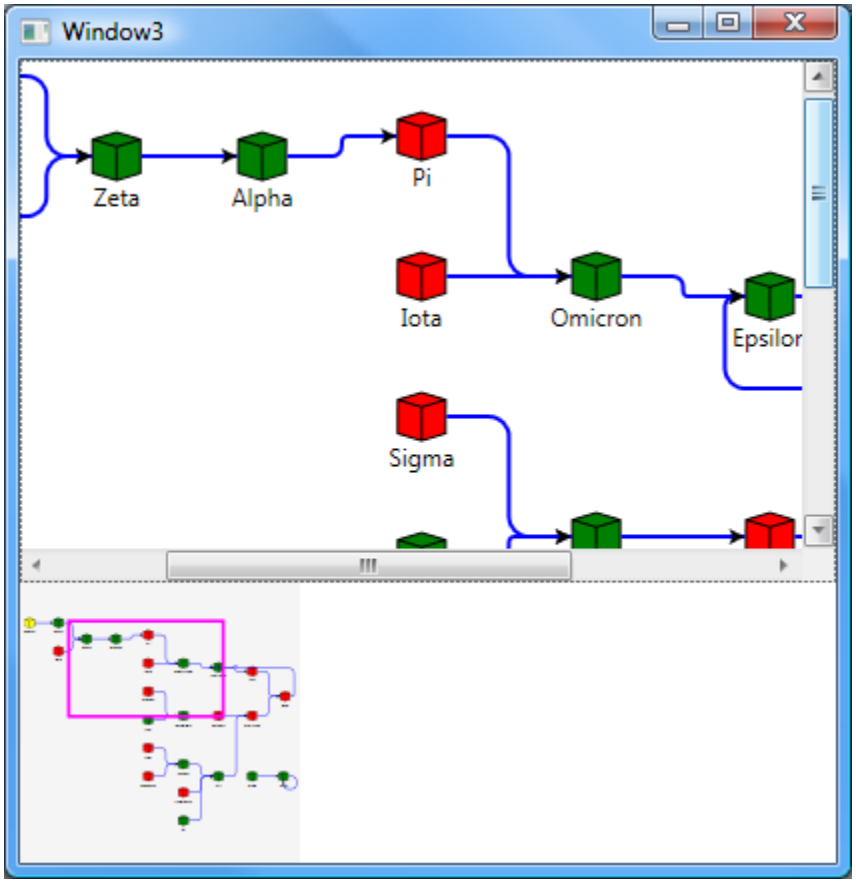
GoXam also provides a specialized kind of **Diagram** called the **Overview**. It displays the whole model shown by another **Diagram** and shows that diagram's viewport. The user can click or drag in the **Overview** to scroll the other diagram's viewport.

```
<Grid>
  <Grid.RowDefinitions>
    <RowDefinition Height="*" />
    <RowDefinition Height="Auto" />
  </Grid.RowDefinitions>
  <go:Diagram x:Name="myDiagram" . . .>
    . . .
  </go:Diagram>
  <go:Overview x:Name="myOverview" Grid.Row="1" HorizontalAlignment="Left"
               Width="140" Height="140" Background="WhiteSmoke" />
</Grid>
```

For the overview to work, the **Overview.Observed** property must be set to refer to the **Diagram** that you want it to show and control. Typically you can do this when the **Diagram** has been initialized:

```
  InitializeComponent();  // inserted by Visual Studio

  this.Loaded += (s,e) => { myOverview.Observed = myDiagram; };
```

This might result in a window such as:



The viewport of `myDiagram` is shown in `myOverview` by the magenta rectangle, which may be dragged by the user. You can customize that rectangle by setting **Overview.BoxTemplate**.

An **Overview** cannot show any unbound parts of the observed diagram – it only shows parts that come from the diagram's model.

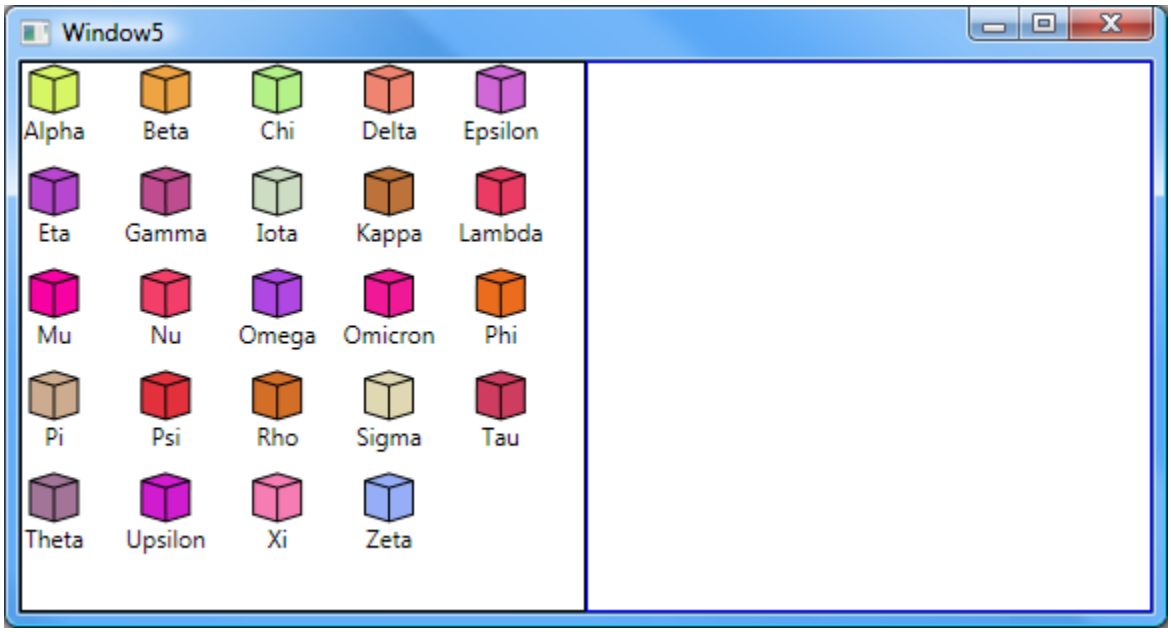
By default an **Overview** uses the same **DataTemplate**s as its observed **Diagram**. Given the reduced scale that the overview shows its parts, this may result in unnecessary overhead, particularly if your nodes use complex templates. It may be much more efficient if you set **Overview.UsesObservedTemplates** to false and define your own simple templates. For your purposes it may be good enough to just use simple colored **Rectangle**s for nodes and simple link shapes without arrowheads or labels for links.

Palette

GoXam also provides a specialized kind of **Diagram** called the **Palette**. It displays a number of nodes in a rectangular grid-like arrangement.

```
<Grid>
  <Grid.ColumnDefinitions>
    <ColumnDefinition Width="*" />
    <ColumnDefinition Width="*" />
  </Grid.ColumnDefinitions>
  <go:Palette Grid.Row="0" Grid.Column="0" x:Name="myPalette"
              BorderBrush="Black" BorderThickness="1"
              NodeTemplate="{StaticResource NodeTemplate}">
  </go:Palette>
  <go:Diagram Grid.Row="0" Grid.Column="1" x:Name="myDiagram"
              AllowDrop="True" BorderBrush="Blue" BorderThickness="1"
              HorizontalContentAlignment="Stretch"
              VerticalContentAlignment="Stretch"
              NodeTemplate="{StaticResource NodeTemplate}"/>
</Grid>
```

Note the attribute `AllowDrop="True"`. This permits drag-and-drops from other controls—in this case from the **Palette** on the left half to the **Diagram** on the right half. To allow the user to start a drag-and-drop from a **Diagram** that can go to other controls, you also need to set `AllowDragOut="True"`; **Palette** sets **AllowDragOut** to true by default.



The corresponding code is:

```
{
  myPalette.Model = new GraphModel<MyData5, String>();
  var letters = new ObservableCollection<MyData5>();
  foreach (String s in new String[] {
        "Lambda", "Mu", "Nu", "Xi",  // note: not in alphabetical order
        "Alpha", "Beta", "Gamma", "Delta", "Epsilon",
        "Zeta", "Eta", "Theta", "Iota", "Kappa",
        "Omicron", "Pi", "Rho", "Sigma", "Tau",
        "Upsilon", "Phi", "Chi", "Psi", "Omega"
      }) {
```

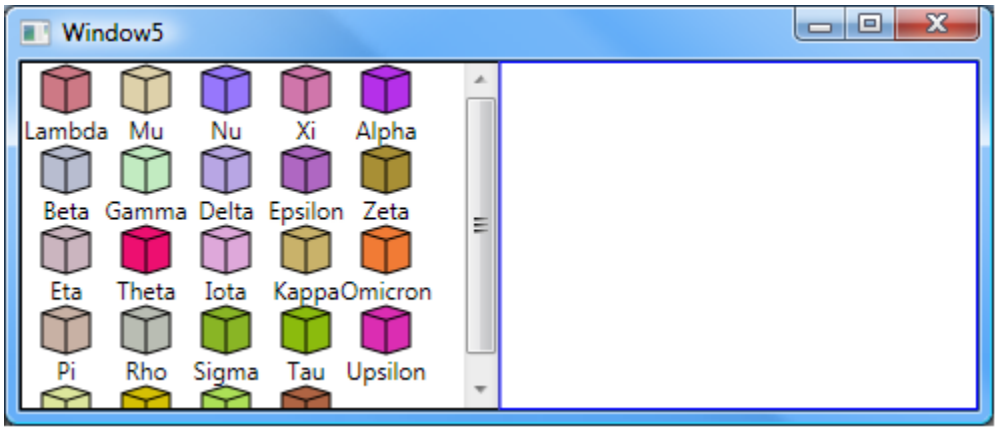
```
    letters.Add(new MyData5() {
      Key=s,
      Color=String.Format("#{0:X}", _Random.Next(0x888888, 0xFFFFFF+1))
    });
  }
  myPalette.Model.NodesSource = letters;

  myDiagram.Model = new GraphModel<MyData5, String>();
  myDiagram.Model.NodesSource = new ObservableCollection<MyData5>();
  myDiagram.Model.Modifiable = true;
}

public class MyData5 : GraphModelNodeData<String> {
  public String Color { get; set; }
}
```

Note also that the nodes have been ordered alphabetically. Sorting normally depends on each node having a **Part.Text** property that is used for sorting alphabetically.

The **Palette** control uses a **GridLayout**. You can customize the layout in the following manner:

```
  <go:Palette . . .>
    <go:Diagram.Layout>
      <go:GridLayout CellSize="10 10" Sorting="Forward" />
    </go:Diagram.Layout>
  </go:Palette>
```

With the explicitly small **CellSize** (smaller than the default, which would be big enough to hold any of the nodes), the nodes are arranged much closer together without overlapping. The **GridSorting.Forward** value for **GridLayout.Sorting** means that the nodes are laid out in the same order in which they were given as the model's **NodesSource**.



As with the other layouts, there are other properties that you may find useful, such as **GridLayout.WrappingColumn** in order to specify the maximum number of columns in the layout.

You can experiment with the various **GridLayout** properties in the GLayout sample of the Demo.

Because **Palette** is a **Diagram**-inheriting class, you can specify your own **DataTemplate** for your nodes. You might want them to be smaller and simpler than in your regular **Diagram**.

Template Dictionaries

The previous examples have shown how to use different **DataTemplate**s for **Node**s, for **Group**s, and for **Link**s. They have also shown how to customize their appearance by binding various property values to your data. But what if you want to have nodes with drastically different appearance or behavior in a single diagram at the same time?

Such capability is supported by having each **Diagram** use a **DataTemplateDictionary** for each type of **Part**: **Node**, **Group**, and **Link**. Each **DataTemplateDictionary** associates a category name with a **DataTemplate**. The **Diagram**'s **PartManager**, which is responsible for creating and removing **Node**s and **Link**s corresponding to the data in the diagram's model, asks the model for a part's category, and then looks it up in the diagram's appropriate dictionary: **Diagram.NodeTemplateDictionary**, **Diagram.GroupTemplateDictionary**, and **Diagram.LinkTemplateDictionary**.

Because a **DataTemplateDictionary** may be shared between diagrams, you can define them as resources in XAML:

```
<go:DataTemplateDictionary x:Key="NodeTemplates">
  <DataTemplate x:Key="">  <!-- the default case: category is empty string -->
    . . .
  </DataTemplate>
  <DataTemplate x:Key="Start">  <!-- when the node's category is "Start" -->
    . . .
  </DataTemplate>
  <DataTemplate x:Key="End">  <!-- when the node's category is "End" -->
    . . .
  </DataTemplate>
</go:DataTemplateDictionary>
```

There are three predefined categories: "" (that's an empty string, the default category), "Comment", and "LinkLabel". For each predefined category you can set the corresponding property via an **DataTemplateDictionary** attribute: **Default**, **Comment**, and **LinkLabel**. For all other categories, you'll will need to add them to the **DataTemplateDictionary** in XAML or by code.

```
<DataTemplate x:Key="NormalNodeTemplate">
    . . .
</DataTemplate>

<go:DataTemplateDictionary x:Key="NodeTemplates"
                           Default="{StaticResource NormalNodeTemplate}">
  <DataTemplate x:Key="Start">
    . . .
  </DataTemplate>
  <DataTemplate x:Key="End">
    . . .
  </DataTemplate>
```

```
  </go:DataTemplateDictionary>
```

In code you can use the **Diagram.FindResource<T>** static method to find a **DataTemplate** that you have defined as a resource in XAML. But remember that these dictionaries might be shared between diagrams. This is particularly true if you create the **DataTemplateDictionary** in XAML as the value of a **Style Setter**.

How does one get the category for a node? You could implement a property on your data class, and set the **…Model.NodeCategoryPath** property to the name of that property. Or you could define a custom **PartManager** and override **PartManager.FindCategoryForNode:**

```
public class CustomPartManager : PartManager {
  protected override String FindCategoryForNode(Object nodedata,
                        IDiagramModel model, bool isgroup, bool islinklabel) {
    // return the desired category here, perhaps computed dynamically
    return ...
  }
}
```

Install this custom **PartManager** with:

```
<go:Diagram . . .>  <!-- must not set NodeTemplate here -->
  <go:Diagram.PartManager >
    <local:CustomPartManager />
  </go:Diagram.PartManager>
</go:Diagram>
```

If you want to use a **DataTemplateDictionary** you must not set the corresponding template property on the diagram – e.g. **Diagram.NodeTemplate**. That property takes precedence over the default template of the template dictionary – e.g. **DataTemplateDictionary.Default** of the **Diagram.NodeTemplateDictionary**.

You can easily swap out the templates, thereby effecting a dramatic change in appearance of the whole diagram. For example, if you want to show different levels of details as the scale changes:

```
    myDiagram.Panel.ViewportBoundsChanged +=
       new RoutedPropertyChangedEventHandler<Rect>(Panel_ViewportBoundsChanged);


  void Panel_ViewportBoundsChanged(object sender,
                                   RoutedPropertyChangedEventArgs<Rect> e) {
    bool simple = myDiagram.Panel.Scale < 0.6;
    if (simple != this.SimpleTemplates) {
      this.SimpleTemplates = simple;
      if (simple) {
        myDiagram.NodeTemplateDictionary =
           Diagram.FindResource<DataTemplateDictionary>("SimpleNodeTemplates");
      } else {
        myDiagram.NodeTemplateDictionary =
           Diagram.FindResource<DataTemplateDictionary>("NormalNodeTemplates");
```

```
      }
    }
  }
  private bool SimpleTemplates { get; set; }
```

The predefined data classes include a **Category** property. Setting this **Category** property after the **Node** (or **Link**) has already been realized will result in that **Part** being removed from the **Diagram** and then re-created using the appropriate new **DataTemplate**.

Generating Images

You can grab part (or all) of the diagram as an image by calling **DiagramPanel.MakeBitmap**.

For example, you could do something like:

```
public void WritePngFile(String filename) {
  Rect bounds = myDiagram.Panel.DiagramBounds;
  double w = bounds.Width;
  double h = bounds.Height;
  // calculate scale so maximum width or height is 2000:
  double s = 1.0;
  if (w > 2000) s = 2000/w;
  if (h > 2000) s = Math.Min(s, 2000/h);
  w = Math.Ceiling(w * s);
  h = Math.Ceiling(h * s);
  BitmapSource bmp = myDiagram.Panel.MakeBitmap(new Size(w, h), 96,
                                  new Point(bounds.X, bounds.Y), s);
  PngBitmapEncoder png = new PngBitmapEncoder();
  png.Frames.Add(BitmapFrame.Create(bmp));
  using (System.IO.Stream stream = System.IO.File.Create(filename)) {
    png.Save(stream);
  }
}
```

The rendering of the **BitmapSource** may be performed asynchronously. If you want to write out a file or do other operations with the bitmap data, you need to do so after the rendering is finished. You can do that by doing that work in an **Action** that you pass as the last argument to **MakeBitmap**:

```
private void Button_Click(object sender, RoutedEventArgs e) {
  Rect bounds = myDiagram.Panel.DiagramBounds;
  double w = bounds.Width;
  double h = bounds.Height;
  // calculate scale so maximum width or height is 500:
  double s = 1.0;
  if (w > 500) s = 500/w;
  if (h > 500) s = Math.Min(s, 500/h);
  w = Math.Ceiling(w * s);
  h = Math.Ceiling(h * s);
  myDiagram.Panel.MakeBitmap(new Size(w, h), 96,
      new Point(bounds.X, bounds.Y), s,
      bmp => {
```

```
            // use the rendered BitmapSource data ...
        });
    }
```

Saving and Loading data using XML

GoXam does not require any particular format or medium for storing diagrams. In many cases the application already has its own database schema or binary file format or whatever, so a good Control should not impose any storage requirements.

The persistence and communication of diagrams should be achieved by storing and loading the model data, not **FrameworkElement**s such as the **Diagram** itself. Because the model data might well be existing application classes about which the model may have limited knowledge and even less control, GoXam does not have a standard storage schema or file format.

This leaves the burden of implementing the most appropriate application-specific persistence mechanisms and protocols to you, the programmer.

However, if you don't mind using XML as the document format, and if you don't have any particular XML schema to which you must adhere, and if you use our predefined model data classes, we do make it easy to save and load your model data. (The predefined model data classes include **GraphLinksModelNodeData** and **GraphLinksModelLinkData**, **GraphModelNodeData**, and **TreeModelNodeData**.)

The model classes include generic methods named **Save** and **Load**. These methods are type-parameterized by the node data type (and in the case of **GraphLinksModel**, by the link data type too). These methods depend on Linq for XML, which you can reference from the System.Xml.Linq assembly and namespace.

Save generates a Linq for XML **XElement**. Once you get the **XElement** you can use its methods to write the XML to a stream or file or string. As you can see in the `Save_Click` button event handler in the samples, it is not too difficult to use:

```
var model = myDiagram.Model as
                GraphLinksModel<MyNodeData, String, String, MyLinkData>;
if (model == null) return;
XElement root =
  model.Save<MyNodeData, MyLinkData>("FlowChart", "MyNodeData", "MyLinkData");
Demo.MainPage.Instance.SavedXML = root.ToString();
```

In the samples the XML is saved as a string in a **TextBox**, where the user might edit it and then reload it.

Load consumes an **XElement** to construct the needed node data and replace the model's **NodesSource**. Here's the main part of the `Load_Click` button event handler, which builds an

XElement from the saved XML string by calling **XElement.Parse**, and then passes the **XElement** to the **Load** method:

```
  var model = myDiagram.Model as
                    GraphLinksModel<MyNodeData, String, String, MyLinkData>;
  if (model == null) return;
  XElement root = XElement.Parse(Demo.MainPage.Instance.SavedXML);
  model.Load<MyNodeData, MyLinkData>(root, "MyNodeData", "MyLinkData");
```

The quoted names (e.g. `"MyNodeData"`) are the names of the XML elements, which in this case happens to be the same as the name of the data types, although that is not required.

Load, like every good method that modifies a model, performs its side-effects within a model transaction.

Adding Data Properties

The code shown above to save and load a model is sufficient when the data classes do not have any additional properties defined on them. But typically you will define some extra properties on your data classes that you will want to be persisted.

You can achieve that by overriding the **MakeXElement** and **LoadFromXElement** methods. In **MakeXElement** you should call the base method to get an **XElement** holding all of the standard information, add your own attributes and child elements as needed, and then return the augmented **XElement**. In **LoadFromXElement** you should call the base method and then set your additional data properties based on information that you read from the given **XElement**.

For example, consider the following node data class:

```
public class GateData : GraphLinksModelNodeData<String> {
  // the type of gate that this node represents; only set on initialization
  public String GateType { get; set; }

  // the shape of the node figure is bound to this property;
  // only set on initialization
  public NodeFigure Figure { get; set; }

  // the current value of this gate; the node color is bound to
  // its Data's Value property and must update with it
  public Boolean Value {
    get { return _Value; }
    set {
      Boolean old = _Value;
      if (old != value) {
        _Value = value;
        RaisePropertyChanged("Value", old, value);
      }
    }
  }
  private Boolean _Value;
```

```
...}
```

This class defines three additional properties. Two properties are not expected to be modified and thus do not implement property change notification, but all three properties are expected to be saved and restored.

Here are the two overridden methods:

```
// support standard reading/writing via Linq for XML
public override XElement MakeXElement(XName n) {
  XElement e = base.MakeXElement(n);
  e.Add(XHelper.Attribute("GateType", this.GateType, ""));
  e.Add(XHelper.AttributeEnum<NodeFigure>(
                          "Figure", this.Figure, NodeFigure.Rectangle));
  e.Add(XHelper.Attribute("Value", this.Value, false));
  return e;
}

public override void LoadFromXElement(XElement e) {
  base.LoadFromXElement(e);
  this.GateType = XHelper.Read("GateType", e, "");
  this.Figure   =  XHelper.ReadEnum<NodeFigure>(
                              "Figure", e, NodeFigure.Rectangle);
  this.Value    =  XHelper.Read("Value", e, false);
}
```

These definitions make use of static methods on the **XHelper** class to simplify your code.

The **XHelper.Attribute** methods are overloaded to handle a number of common data types for properties. The principal purpose of these helper methods is to avoid writing out attributes or elements when the current value is equal to the default value. As you can see when you examine the resulting XML, it is much more concise when attributes with the default value are not generated.

The **XHelper.Read** methods are also overloaded to handle common data types. Their principal purpose is to handle the cases when either the **XElement** is not present, or more likely, that the **XAttribute** is not present. The **Read** methods also handle incorrectly formatted input that causes exceptions. In all of these cases, the **Read** method will return the default value for that property. Remember that if the type is a **Double** the default value that you pass to **XHelper.Read** needs to be a double too, not an integer:

```
this.Width = XHelper.Read("Width", e, 80.0); // must include ".0"
```

When initializing a model you do not have to perform all model side-effects within a model transaction. However, when modifying an existing model, you should do so. For example:

```
private void Button_Click(object sender, RoutedEventArgs e) {
  myDiagram.StartTransaction("Add Stuff");
```

```
    // create some nodes and add them to the model's NodesSource collection
    . . .

    myDiagram.CommitTransaction("Add Stuff");
  }
```

Your application may choose to save properties such as the points in a **Link**'s **Route**. Although there are **XHelper** methods for reading/writing a list of **Point**s, and there is a **GraphLinksModelLinkData.Points** property, at the current time one cannot data-bind the **Link.Route.Points** property to the **Points** property on the data. Instead you will need to explicitly copy the route points to and from the data. This is demonstrated in the StateChart sample in the demo app.

However, there is an additional complication when loading link route points. The actual construction of the nodes, which finishes asynchronously after your loading code executes, will naturally cause all connected links to be re-routed. It is only afterwards that you can copy the points from the data to the **Link.Route**. The StateChart sample demonstrates how to do that, by implementing a **LayoutCompleted** event handler that is executed only once per load.

Updating a database

Unlike the save and load model typically used when writing and reading XML, updating a database wants to occur frequently, operating only on the data that has changed. The usual strategy is to update the database when a diagram transaction completes. The way to do that is to implement a **DiagramModel.Changed** event handler:

```
    model.Changed += model_Changed;
```

(The event handler could also be in an override of **DiagramModel.OnChanged** or in an override of **PartManager.OnModelChanged**; remember to call the base method first!)

```
void model_Changed(object sender, ModelChangedEventArgs e) {
  if (e.Change == ModelChange.CommittedTransaction) {
    var cedit = e.OldValue as UndoManager.CompoundEdit;
    if (cedit == null) return;

    // maybe wrap the following code in a database transaction:

    // iterate over the changes within the transaction
    foreach (IUndoableEdit edit in cedit.Edits) {
      var ed = edit as ModelChangedEventArgs;
      if (ed == null) continue;
      switch (ed.Change) {
        case ModelChange.AddedNode:
          // add a record describing a node data (e.Data)
          break;
        case ModelChange.RemovedNode:
          // remove any record corresponding to a node data
          break;
```

```
          // handle other ModelChange cases
        }
      }
    }
  }
```

The **CompoundEdit** contains a list of all of the **ModelChangedEventArgs.** Each describes a change to the model that occurred within that transaction. The above code shows checking for any additions or removals of node data from the model. The UpdateDemo sample of the Demo app demonstrates a similar technique, although for a different purpose.

If your application supports undo and redo, you will also want to catch **ModelChange.FinishedUndo** and **FinishedRedo**.

```
if (e.Change == ModelChange.FinishedUndo) {
  var cedit = e.Data as UndoManager.CompoundEdit;
  if (cedit == null) return;
  // NOTE: operate on edits in reverse order for Undo
  foreach (IUndoableEdit edit in cedit.Edits.Reverse()) {
    . . .
  }
} else if (e.Change == ModelChange.FinishedRedo) {
  var cedit = e.Data as UndoManager.CompoundEdit;
  if (cedit == null) return;
  // but operate on edits in forwards order for Redo
  foreach (IUndoableEdit edit in cedit.Edits) {
    . . .
  }
}
```

Deploying your application

Before you can distribute your application, you will need to purchase a developer's license for GoWPF, install a development license key on your development machine, and insert a run-time license key assignment statement into your application constructor. The **GoXam License Manager** will help you perform these steps. You can find this application in the Start Menu or in the **bin** subfolder of the installed kit in your **Documents** folder.

You can purchase a development license at our web site:
http://www.nwoods.com/sales/ordering.htm.
You will be sent an acknowledgement e-mail that includes your order number. Please make sure that you will always receive e-mail from nwoods.com.

Once you have your order number, run the **GoXam License Manager** on your development machine and click on the "Request a License" button. This will take you to our web site where you can enter your e-mail address, the order number, and which component your application is using. You will be sent an e-mail containing a license key for your particular development machine.

If you are unable to connect your development machine to the internet, you can go to: http://www.nwoods.com/activate.asp?sku=xam and enter the same information plus the name of your login account and the name of your development machine.

Once you get the license key, click on the "Install a License" button in the **GoXam License Manager**. Copy the license key, which is shown in bold in the license key e-mail message, into the License Manager's text box. If you pasted a valid license key, the "Store into Registry" button becomes visible and you can click it.

When your machine has a development license installed, you can generate run-time licenses at your convenience. Each application that you want to distribute will need a separate run-time license. Start the **GoXam License Manager** and click on the "Deploy an Application" button.

Because the run-time license is tied to the name of the application, you will need to enter the application name. This is the name of your managed EXE or DLL (without a directory path or the file type) that contains the definition of your application class inheriting from **System.Windows.Application**. It *is not* the name of your project nor is it a namespace nor is it the name of your main window. It *is* the name of the assembly that defines the class of the value of **Application.Current**.

If you are building a Windows Forms app or are deploying a DLL into an unmanaged application, there might not be a value for the static/shared property **System.Windows.Application.Current.** If that is the case, you can just create an instance of a trivial class that inherits from **System.Windows.Application**; creating it will automatically set **Application.Current**.

You can then paste the code from the clipboard into the constructor for your application. The constructor is usually in the code file `App.xaml.cs`. It will look something like:

```
// This is a license for Northwoods.GoWPF version 1.0 (or earlier)
// Put the following statement in your MyApplication application constructor:
Northwoods.GoXam.Diagram.LicenseKey =
  "f9fWgfq7Q3F7xU4QGRTUUMbhBv3lFOdPfCOkOdmvoldwOe9Y1vmhIM3ClYRcOOIb+yLhoN+Y+D2x5RlEbSZ+N0nZo=";
```

After you re-compile your application, you should be able to deploy it successfully to as many machines as you wish.

Remember to generate and use a new run-time license key if you change the name of your application assembly or if you upgrade to a newer major or minor version of the GoXam DLL. The baselevel version number does not matter, so for example, upgrading a DLL from *m*.*n*.1 to *m*.*n*.2 can be performed at any time using the same license key.

Appendix

These are additional topics of lesser importance for most applications.

Diagram Templates

Because **Diagram** is a regular **Control**, you can change the overall appearance by replacing its **Template**. This will not affect the appearance of any node or link.

The default **ControlTemplate**, when simplified, is:

```
<ControlTemplate TargetType="go:Diagram">
  <Border Background="{TemplateBinding Background}"
          BorderBrush="{TemplateBinding BorderBrush}"
          BorderThickness="{TemplateBinding BorderThickness}">
    <ScrollViewer HorizontalScrollBarVisibility="Auto"
                  VerticalScrollBarVisibility="Auto"
                  CanContentScroll="True">
      <go:DiagramPanel x:Name="Panel"
        Stretch="{TemplateBinding Stretch}"
        Padding="{TemplateBinding Padding}"
        HorizontalContentAlignment="{TemplateBinding HorizontalContentAlignment}"
        VerticalContentAlignment="{TemplateBinding VerticalContentAlignment}" />
    </ScrollViewer>
  </Border>
</ControlTemplate>
```

If there is no **DiagramPanel** in the **ControlTemplate**, the diagram won't be useful.

Layers

A **Layer** holds a collection of parts that are drawn in (approximately) the same Z-order position. There are two kinds of layers: **NodeLayer** and **LinkLayer**. A **NodeLayer** can hold **Group**s and **Adornment**s as well as regular **Node**s.

Initially each **DiagramPanel** creates nine layers. These are:

- "Background" **NodeLayer**
- "Background" **LinkLayer**
- "", the default **NodeLayer**
- "", the default **LinkLayer**
- "Foreground" **NodeLayer**
- "Foreground" **LinkLayer**
- "Tool" **NodeLayer**, a layer used by tools for temporary nodes
- "Tool" **LinkLayer**, a layer used by tools for temporary links
- "Adornment" **NodeLayer**, a layer used for **Adornment**s such as selection handles

You can specify which layer a **Part** should be in by setting the **Part.LayerName** attached property. But you can also remember the layer in your data. For example, if you add a "Layer" property to your node data:

```
public class MyData : GraphLinksModelNodeData<String> {
  public String Layer {
    get { return _Layer; }
    set {
      if (_Layer != value) {
        String old = _Layer;
        _Layer = value;
        RaisePropertyChanged("Layer", old, value);
      }
    }
  }
  private String _Layer = "";  // the default layer name is an empty string
}
```

You will then be able to bind the attached **Part.LayerName** property to that **MyData.Layer** property:

```
<DataTemplate x:Key="NodeTemplate1">
  <Border go:Node.Location="{Binding Path=Data.Location, Mode=TwoWay}"
          go:Node.LayerName="{Binding Path=Data.Layer}"
          . . .>
    . . .
  </Border>
</DataTemplate>
```

Because the **MyData.Layer** property setter calls **RaisePropertyChanged**, if you set this data property, it will automatically change the diagram layer that the node is in.

Another common usage of layers is to temporarily bring parts forward or push them back. For example, if you want a selected node to be brought to the "Foreground" layer, you can bind the **Part.LayerName** to whether the node is selected, using a converter defined as a resource:

```
  <go:BooleanStringConverter x:Key="theBooleanLayerConverter"
                             TrueString="Foreground" FalseString="" />
```

This converter will return "Foreground" when the input value is true, and will return the empty string otherwise. In this case there is no need for any layer information in your data class.

```
<DataTemplate x:Key="NodeTemplate2">
  <!-- note that the binding path is Path=Node... not Path=Data... -->
  <Border go:Part.LayerName="{Binding Path=Node.IsSelected,
                      Converter={StaticResource theBooleanLayerConverter}}"
        . . .>
    . . .
  </Border>
</DataTemplate>
```

Note that the binding path is not to a property on the node's **Data**, but to a property (**Node.IsSelected**) on the node itself.

A more general solution involving per-node layer information on the data is possible by using triggers:

```
<DataTemplate x:Key="NodeTemplate3">
  <Border . . .>
    <Border.Style>
      <Style>
        <Setter Property="go:Part.LayerName"
                Value="{Binding Path=Data.Layer}" />
        <Style.Triggers>
          <Trigger Property="Part.IsSelected" Value="True">
            <Setter Property="go:Part.LayerName" Value="Foreground" />
          </Trigger>
        </Style.Triggers>
      </Style>
    </Border.Style>
    . . .
  </Border>
</DataTemplate>
```

The advantage to using triggers here is that the value of **Part.LayerName** will automatically be restored to its originally bound value, whatever it was, once the part is no longer selected.

Decorative Elements and Unbound Nodes

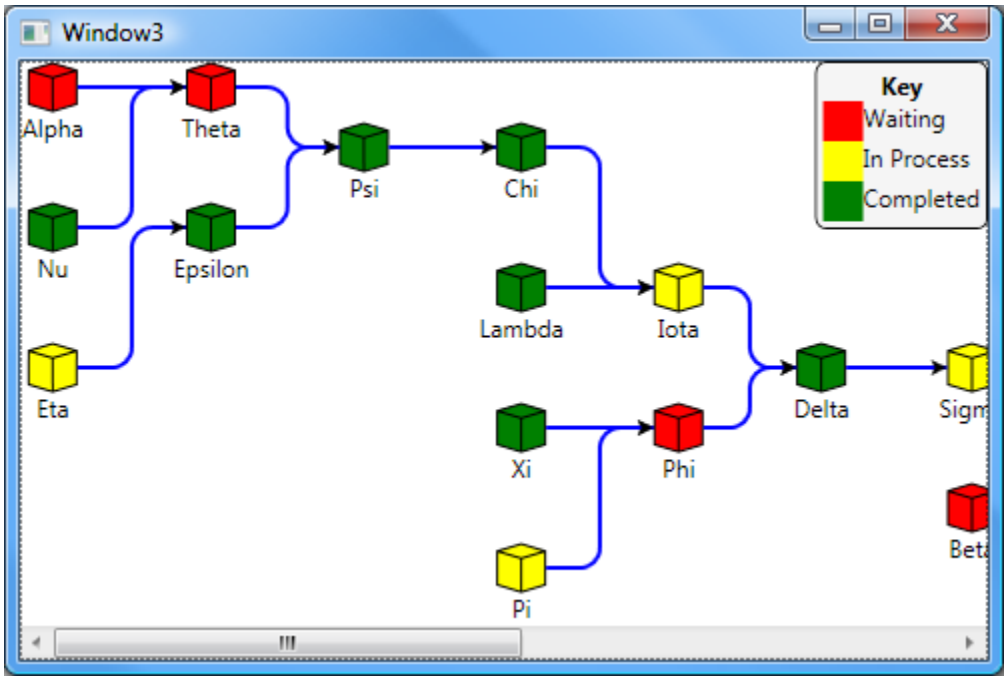
Most of the previous examples have made use of nodes and links that were bound to data held by the diagram's model. This is the normal scenario – the presentation/appearance of the data is separated from the implementation of the data and the rest of your application. The appearance of each part is defined by XAML as part of **DataTemplate**s that are applied (copied) and bound to the data.

However there are times when you may wish to create visual objects that are not bound to any application data. There are two categories for such visual objects: ones that have a fixed position within the diagram's viewport, and ones that scroll and zoom along with the regular nodes and links of the diagram.

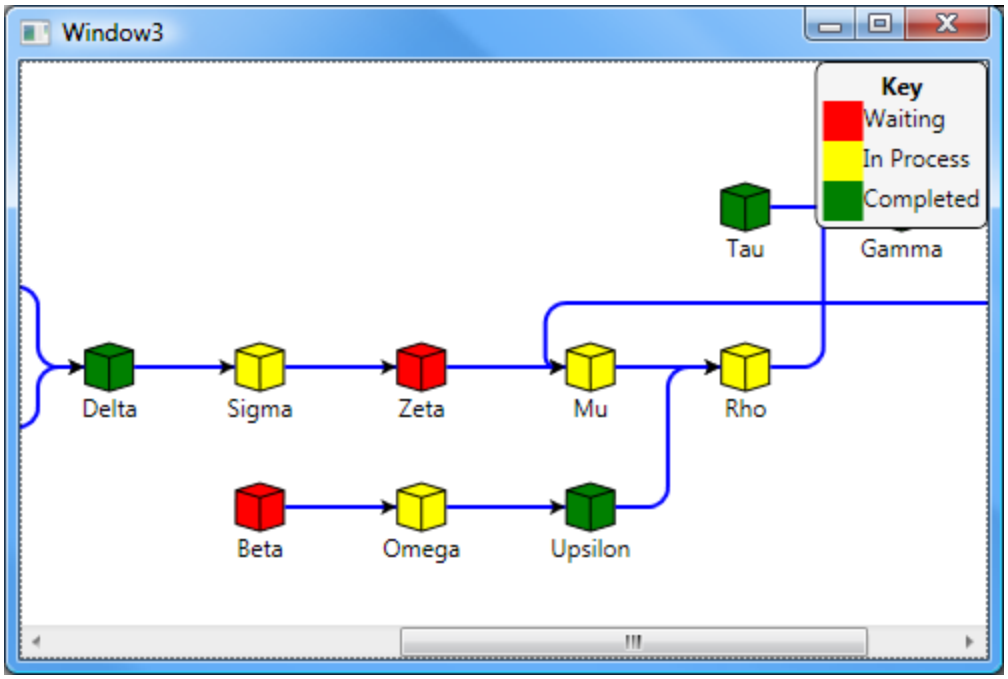
You can implement objects in the former category as regular **FrameworkElement**s outside of the **Diagram** but overlapping it. For example, you may want to create a legend describing parts of your diagram. You could define the legend as regular XAML that is not part of a **DataTemplate** but is part of your presentation along with the **Diagram** itself:

```
<Grid>
  <go:Diagram . . . >
  </go:Diagram>
  <!-- Both elements occupy the same (and the only) cell of the Grid -->
  <!-- Define the legend in another Grid surrounded by a Border -->
```

```
  <Border BorderBrush="Black" BorderThickness="1" Background="WhiteSmoke"
          CornerRadius="5" Padding="3"
          HorizontalAlignment="Right" VerticalAlignment="Top">
    <Grid>
      <Grid.ColumnDefinitions>
        <ColumnDefinition Width="Auto" />
        <ColumnDefinition Width="Auto" />
      </Grid.ColumnDefinitions>
      <Grid.RowDefinitions>
        <RowDefinition Height="Auto" />
        <RowDefinition Height="Auto" />
        <RowDefinition Height="Auto" />
        <RowDefinition Height="Auto" />
      </Grid.RowDefinitions>
      <TextBlock Grid.Row="0" Grid.Column="0" Grid.ColumnSpan="2"
                 Text="Key" FontWeight="Bold" HorizontalAlignment="Center" />
      <Rectangle Grid.Row="1" Grid.Column="0"
                 Fill="Red" Width="20" Height="20" />
      <TextBlock Grid.Row="1" Grid.Column="1"
                 Text="Waiting" />
      <Rectangle Grid.Row="2" Grid.Column="0"
                 Fill="Yellow" Width="20" Height="20" />
      <TextBlock Grid.Row="2" Grid.Column="1"
                 Text="In Process" />
      <Rectangle Grid.Row="3" Grid.Column="0"
                 Fill="Green" Width="20" Height="20" />
      <TextBlock Grid.Row="3" Grid.Column="1"
                 Text="Completed" />
    </Grid>
  </Border>
</Grid>
```



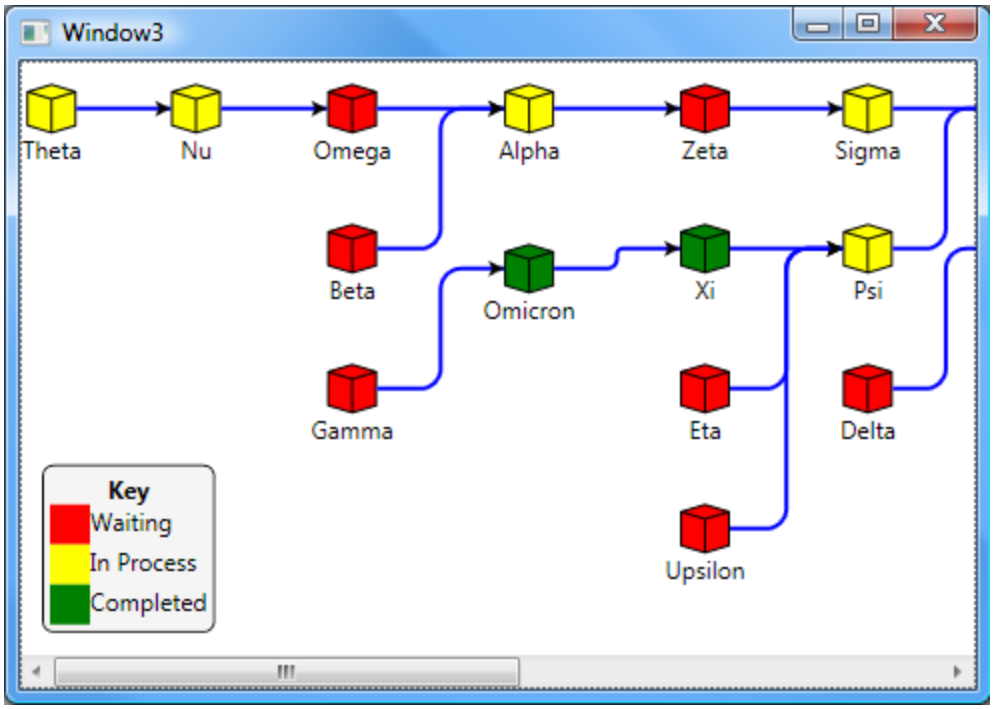
When the user scrolls to the right, the legend stays in place, because it is not part of the diagram:



But if you move that legend XAML into a **Node** that is a child of your **Diagram** element, the legend will be a node that will be scrolled and zoomed along with the rest of your diagram.

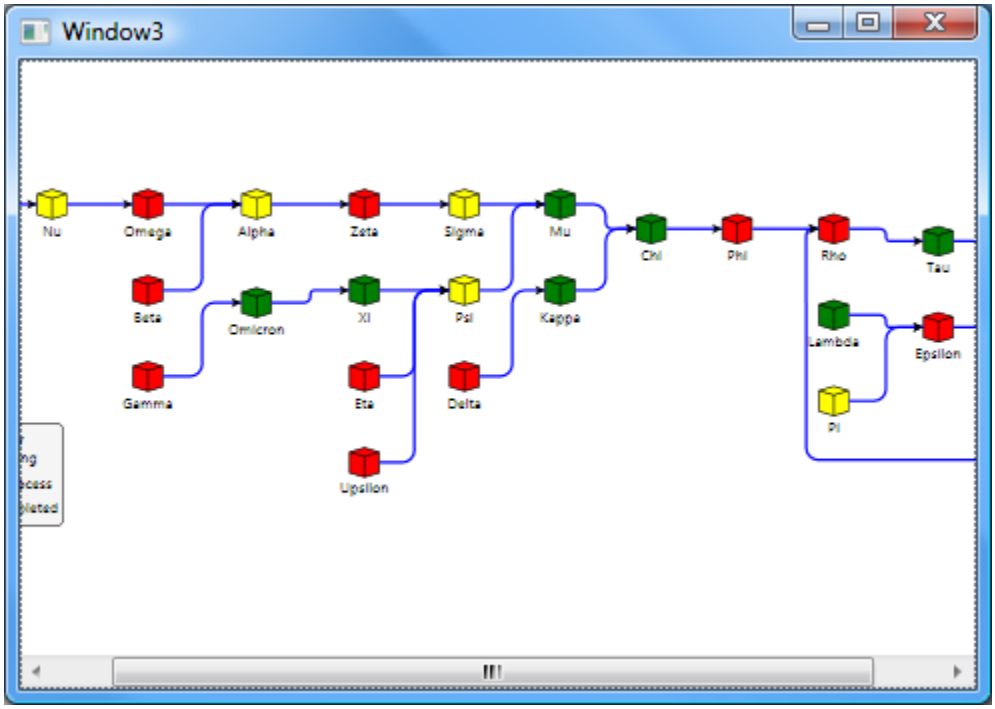
```
<Grid>
  <go:Diagram . . . >
    <go:Node>
      <Border BorderBrush="Black" BorderThickness="1" Background="WhiteSmoke"
              CornerRadius="5" Padding="3"
              go:Part.Selectable="False" go:Node.Location="20 200">
        <Grid>
          . . . same legend as before . . .
        </Grid>
      </Border>
    </go:Node>
  </go:Diagram>
</Grid>
```

Note that we have explicitly set the **Location** for this legend **Node.** Also, to keep users from playing with the legend, you may want to make the legend not **Selectable**.



Now when you scroll the diagram a bit and zoom out a little, you can see that the legend stays in the same place relative to all of the other nodes and links. In the screenshot that follows, the legend has been scrolled partly out of view on the left.

This kind of **Node** is not bound to model data, so we call it an “unbound” node.



Unbound Links

Unbound **Node**s can also be connected by unbound **Link**s. These unbound **Part**s are added to the **Diagram.InitialParts** collection, after which the diagram adds them to its **PartsModel**, which is a special model meant to hold these unbound nodes and links.

In order to create such graphs, you can set the **Node.Id** property and refer to those nodes by setting the **Link.PartsModelFromNode** and **Link.PartsModelToNode** properties. These are regular dependency properties on **Node** and **Link**, not attached properties that are set on the root visual element of each **Node** and **Link**, so they cannot be bound to node or link data properties.

```
<go:Diagram  . . .>
  <!-- this Id is for the PartsModel holding these initial Parts -->
  <go:Node Id="Node1">
    <!-- but the rest of the XAML is the same as in a DataTemplate -->
    <StackPanel go:Node.Location="10 10" go:Part.SelectionAdorned="True">
      <Rectangle Width="30" Height="30" Fill="Blue" HorizontalAlignment="Center"
                 go:Node.PortId="" />
      <TextBlock Text="Node1" HorizontalAlignment="Center" />
    </StackPanel>
  </go:Node>
  <go:Node Id="Node2">
    <StackPanel go:Node.Location="110 10" go:Part.SelectionAdorned="True">
      <Ellipse Width="30" Height="30" Fill="Green" HorizontalAlignment="Center"
               go:Node.PortId="" />
      <TextBlock Text="Node2" HorizontalAlignment="Center" />
    </StackPanel>
  </go:Node>
  <!-- these two Link properties are for the PartsModel -->
  <go:Link PartsModelFromNode="Node1" PartsModelToNode="Node2">
    <!-- but the rest of the XAML is the same as in a DataTemplate -->
    <go:LinkPanel go:Part.SelectionAdorned="True"
                  go:Part.SelectionElementName="Path">
      <go:LinkShape Name="Path" Stroke="DarkMagenta" StrokeThickness="2" />
      <!-- "to" arrowhead -->
      <Polygon Fill="DarkMagenta" Points="8 4  0 8  2 4  0 0"
               go:LinkPanel.Index="-1" go:LinkPanel.Alignment="1 0.5"
               go:LinkPanel.Orientation="Along" />
      <!-- "from" arrowhead -->
      <Polyline Fill="White" Stroke="DarkMagenta" StrokeThickness="2"
                Points="0 0  8 4  0 8"
                go:LinkPanel.Index="0" go:LinkPanel.Alignment="0 0.5"
                go:LinkPanel.Orientation="Along" />
    </go:LinkPanel>
  </go:Link>
</go:Diagram>
```

This produces:

For group membership, you can set the **Node.PartsModelContainingSubgraph** property.

Caution: although the **Node**s and **Link**s that you implement in XAML as the initial parts for a **Diagram** can be selected and dragged, at the current time they cannot be copied nor can its operations be undone/redone. Normally you will make each **Part** not **Selectable**, or you will add them to the "Background" **Layer** and set **Layer.AllowSelect** to false.

Performance considerations

With small to medium size graphs, the load time and the interactive performance should be good. However, as your diagram gets to thousands of **Node**s and **Link**s, you will find that the load time may start to get long. Interactive performance should remain reasonable as long as the user does not zoom out so that most of the nodes are visible.

The load time is due to the overhead of creating all of the **FrameworkElement**s of the **DataTemplate**s. You can improve this initialization performance by keeping your **DataTemplate**s as simple as possible. Reduce how many elements you use. Avoid using **Control**s. Avoid creating **FrameworkElement**s that are not visible. Minimize the number of data-bindings too.

Interactive performance, typically for moving nodes, is reduced when the link **Route**s have **Routing="AvoidsNodes"** and/or when **Curve="JumpOver"** or **"JumpGap"**.

Some diagram layouts are faster than others. **TreeLayout** is much faster and scales up much better than **LayeredDigraphLayout**.

If you intend to handle many thousands of nodes and links, we recommend that you experiment. When showing such large diagrams your users may find it easier to comprehend when you use progressive disclosure or drill-down. The use of expand and collapse, for both subtrees and subgraphs, can help greatly too. Look at the IncrementalTree and Grouping samples in the demo.

If you have many thousands of nodes and links and you know ahead-of-time exactly where all of the nodes are located, you may be able to use the virtualization technique shown in the Virtualizing sample of the demo. The sample also can support **Group**s. But you will need to adapt the code in order for the virtualization to work successfully. The Virtualizing sample creates 62500 nodes and 62499 links. Initialization time is relatively good, but scrolling and zooming is slower than when not virtualized.

Of course virtualization doesn't help if the diagram has to show all of the nodes anyway, either because they are all positioned very near to each other or because the diagram has been scaled down to show most or all of the **DiagramPanel.DiagramBounds**. That's the expectation for the **Overview** control. If you can't avoid using an **Overview**, customize the **DataTemplate**s used by the **Overview** to be absolutely as simple as can be. This is usually easy to implement because the nodes will be so small that a simple rectangle or ellipse is sufficient.

Finally, if you really need to deal with tens to hundreds of thousands of nodes, you may want to use GoDiagram instead. You can easily handle tens of thousands of nodes in GoDiagram for Windows Forms. Only layered-digraph layout is a serious limitation. But it requires much more coding to produce complex or pretty nodes and to integrate with data. In those respects using XAML in GoXam is much superior to GoDiagram.