GoDiagram for .NET

Interactive Diagram Classes

User Guide

This guide provides information on using:

GoDiagram[TM] **for Microsoft® .NET Windows Forms (GoDiagram Win)**

GoDiagram[TM] **for Microsoft® .NET Compact Framework (GoDiagram Pocket)**

GoDiagram[TM] **for Microsoft® ASP.NET Web Forms (GoDiagram Web)**

Controls and classes for building interactive graphical diagrams

for Windows Forms and ASP.NET.

July 2011

Northwoods Software Corporation
142 Main St.
Nashua, NH 03060 USA

http://www.nwoods.com/go/

CONTENTS

GoDiagram User Guide

Purpose of this guide

This guide provides an overview of **GoDiagram for .NET**, .NET class libraries containing sets of controls for easily building interactive diagrams. **GoDiagram for .NET Windows Forms** is for stand-alone Windows Forms applications, **GoDiagram for ASP.NET Web Forms** is for ASP.NET Web Forms applications, and **GoDiagram for .NET Compact Framework** is for Pocket PC applications.

Understanding this guide requires familiarity with the .NET platform and with Windows Forms and ASP.NET.

The classes in the two libraries are nearly identical—in fact they are all named identically to facilitate learning and porting code. The only differences are those required for participating well in their respective environments, Windows Forms or Web Forms. Some of the differences are discussed in the corresponding introductory document, either `GoWinIntro.doc` or `GoWebIntro.doc`. A listing of all of the differences is maintained in `GoWinWebDiffs.doc`.

For more detailed information about the types, classes and interfaces in **GoDiagram for .NET Windows Forms** (the **Northwoods.Go** library in the `Northwoods.Go.dll` assembly), see the GoDiagram Win Reference Manual, `GoWin.chm`, a compiled HTML archive.

For more detailed information about the types, classes and interfaces in **GoDiagram for .NET Compact Framework** (the **Northwoods.Go** library in the `Northwoods.GoPocket.dll` assembly), see the GoDiagram Win Reference Manual, `GoWin.chm`.

For more detailed information about the types, classes and interfaces in **GoDiagram for ASP.NET Web Forms** (the **Northwoods.GoWeb** library in the `Northwoods.GoWeb.dll` assembly), see the GoDiagram Web Reference Manual, `GoWeb.chm`.

The Frequently Asked Questions list has been moved into a separate document, `GoDiagramFAQ.chm`.

Northwoods Software also maintains a forum on its website:
http://www.nwoods.com/forum.

Terminology

We will often use "GoDiagram" or "Go" to refer to **GoDiagram for .NET**.

Because of the great commonality between the **GoDiagram for .NET Windows Forms** and **GoDiagram for ASP.NET Web Forms** products, "GoDiagram" or "Go" can also refer equally to either product.

The short name for **GoDiagram for .NET Windows Forms** is "GoDiagram Win", or sometimes "GoWin".

The short name for **GoDiagram for .NET Compact Framework** is "GoDiagram Pocket", or sometimes "GoPocket".

Features that are specific to Windows Forms are present in both GoDiagram Win and GoDiagram Pocket.

The short name for **GoDiagram for ASP.NET Web Forms** is "GoDiagram Web", or sometimes "GoWeb". That product is also implied when talking about ASP.NET or about Web Forms.

Note that the same namespace is used for both Windows Forms libraries, **Northwoods.Go**, even though the assembly names are different. A different namespace is used for Web Forms, **Northwoods.GoWeb**.

A "user" is the person who sees the Go controls and displayed objects and who uses the mouse and keyboard to manipulate them.

"You" refers to the programmer who is developing an application using the Go controls. Of course every developer is also a "user", when debugging and testing an application.

1. INTRODUCTION

The Go libraries are sets of controls and classes built on the .NET platform. Go makes it easy to deliver user interfaces that allow users to see and manipulate diagrams of two-dimensional graphical objects arranged in a scrollable window.

Go provides a variety of basic graphical objects such as rectangles, ellipses, polygons, text, images, and lines. You can group objects together to form more complex objects. You can customize their appearances and behaviors by setting properties and overriding methods.

A Go view is a control that displays a Go document. It supports mouse-based object manipulation, including selecting, resizing, moving and copying using drag-and-drop. Go organizes input behaviors into tools that you can modify, override, or add or remove from a view. The view also supports in-place editing, printing, and grids.

A Go document implements a model that supports manipulation of objects. Adding an object to the document makes it visible in the document's views. You can organize objects in layers. Go provides support for composing and manipulating graphs (node & arc diagrams), where nodes have ports that are connected by links.

The Go library is flexible and extensible. Many predefined node classes make it easy to build many kinds of diagrams. You can easily customize most objects for application-specific purposes by setting properties or by subclassing. You can add completely new graphical objects to the existing framework.

Other libraries, named **Northwoods.Go*.***, extend Go by providing automatic layout algorithms, meters/dials/gauges, and support for reading/writing XML. These are documented either in separate manuals or later in this one.

If you are using **GoDiagram Win**, you should read `GoWinIntro.doc` first.

For **GoDiagram Pocket**, please read `GoPocketIntro.doc` first.

For **GoDiagram Web**, please read `GoWebIntro.doc` first.

These documents also discuss licensing and deployment issues.

---

2. GO CONCEPTS

This guide assumes you are already familiar with Windows, the .NET Common Language Runtime and .NET Framework classes, and **System.Windows.Forms** or **System.Web.UI** and **System.Drawing** in particular. Go builds directly on this framework, so understanding them is a prerequisite for understanding Go.

All Go classes follow the convention of using "Go" in their name (e.g., **GoView**) to avoid name conflicts when using the **Northwoods.Go** or **Northwoods.GoWeb** namespace; any other class names in this document are .NET Framework classes (e.g., **System.Drawing.Graphics**) or in the sample source code.

Design Philosophy

Go has been designed for high performance, ease of use and flexibility to meet a wide variety of requirements. You can easily customize Go just by setting properties on views, documents, and objects, or by providing event handlers for the view or documents.

While Go may not provide every last feature you may need, Go does provide many methods that you can call or override to get exactly the behavior you want.

Go is designed to allow you to organize your application in ways that scale up as well as in ways that are expedient. For many actions and events, there are several steps performed by several different classes. Each step can be customized or overridden.

Although you may find that there appear to be multiple ways of doing something, one of those ways is likely to be better than the others for your application. The benefit is that it is easier to put your code where it belongs once you understand how you are likely to need to maintain and extend it in the future.

A common situation is that you might add a particular event handler in your initial implementation in order to achieve certain functionality. Later you realize the code doesn't belong with the form, but with the view, with the document or with the individual object class. We have designed Go to make different application architectures easier to implement.

One way to discriminate between different implementation strategies is to decide if the actions are just associated with the user's interactive direct-manipulation of an object or if the actions should occur programmatically, no matter the reason nor what code is executing. For example, disallowing the user from using the mouse interactively to move an object in a diagram does not prevent some code from changing its position by assigning its position programmatically. However it is also possible to implement objects whose position cannot be changed by any means whatsoever.

If you want to see what the class hierarchy is for Go, you won't find that here. Instead you will find a much more informative and interactive way to see a class hierarchy by running the Classier sample application.

Documents

Go uses a model-view-controller architecture. **GoDocument** serves as the model, i.e. a container providing the abstract representation of the things the user may see in a view.

Documents provide runtime storage for displayable objects. A document is the object that contains the list of layers of graphical objects to be displayed in one or more views. When you want to have a graphical object appear to the user, you create it, make sure it has a reasonable size and position and any other properties you care about, and then add it to a document's layer.

Class **GoDocument** inherits from **System.Object**; i.e., a document and its objects do not depend on the existence of a window. A document has an instance of **GoLayerCollection**, which is a collection of **GoLayer** instances. Each **GoLayer** is a collection of **GoObject**s, which are the things users can see and manipulate in a view.

Each document has a number of properties that affect its appearance and behavior. These include properties such as paper color and whether the user can delete or insert or move or copy objects.

GoDocument supports one event, **Changed**, so that interested observers can be notified of changes to the document or to any of its objects.

Views

GoView serves as the view in the model-view-controller architecture. Views provide a window on the graphical objects stored in a document. A view defines how the user sees the objects and interacts with them. Each view handles its document's **Changed** event so that it can keep its window up-to-date with all of the objects in the document.

Class **GoView** inherits from **System.Windows.Forms.Control** or **System.Web.UI.WebControl**. It has many properties for controlling its appearance and behavior. As a regular control, **GoView** can be placed in a form

designer's toolbox and drag-and-dropped onto a form to be instantiated. It raises many kinds of events involving clicking, selecting, moving, or deleting objects or clicking in the background.

GoView provides end-user editor support for selection, within-the-view drag-and-drop, copy-and-paste, and grid display. GoDiagram Win also provides support for hovering, cross-window drag-and-drop, in-place text editing, and printing.

There are two subclasses of **GoView** that provide more specialized behavior. Here is a class hierarchy diagram of the **GoView** classes:
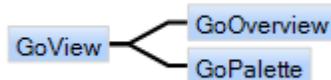
- **GoOverview** presents a reduced-scale view of a different view, along with the ability for the user to pan and zoom that other view by dragging around a rectangle within the overview, and by dragging in the background to specify a new position and scale for that other view.
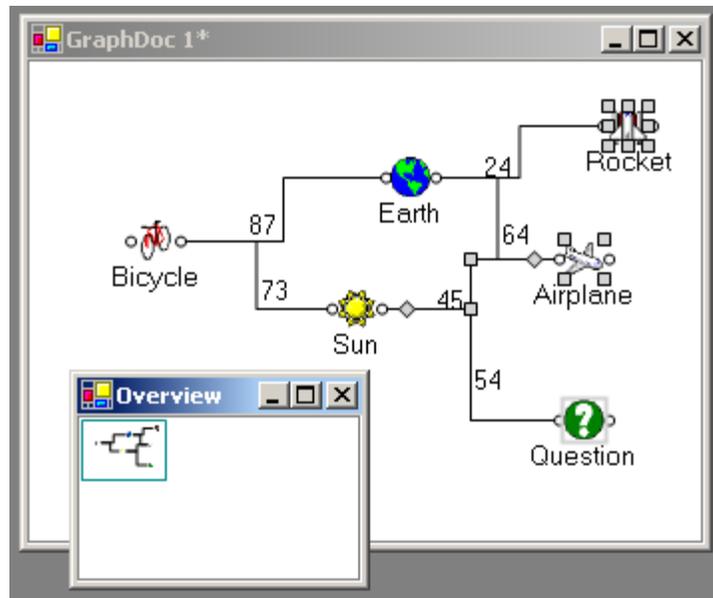
- **GoPalette** holds a read-only collection of objects, laid out in a grid, for the user to select and drag into a **GoView** (except in GoDiagram Pocket).

Tools

As a regular control, **GoView** is responsible for handling all input events. However, to make input handling more flexible and better organized, **GoView** just passes on all regular mouse and keyboard input to one of the objects implementing **IGoTool** that each view maintains. Thus both **GoView** and **GoTool** serve as the controller part of the model-view-controller architecture.

Class **GoTool** is the standard implementation of **IGoTool**, deriving from **System.Object**. It has methods for handling mouse down/move/up and keyboard input. It also has methods that the view can call to control its tools: to determine if that tool instance is applicable and to start and stop it. There are subclasses of **GoTool** for each of the kinds of predefined user interaction, such as **GoToolSelecting**, **GoToolDragging**, **GoToolLinking**, **GoToolResizing**, **GoToolRubberBanding**, **GoToolZooming**, and **GoToolPanning**.

The **GoToolManager** tool is normally the view's default tool; it is responsible for invoking the appropriate specific tool and for handling standard keys such as Escape, PageUp/Down, Delete and Ctrl-X/Ctrl-C/Ctrl-V when no other tool is active.

GoView and the **GoTool**s, in conjunction with the individual graphical objects, provide a default user interaction style that is consistent with standard usability guidelines for selecting, moving, resizing and other user interactions. However, user interactions defined by **GoView**, **GoTool**, **GoDocument** and **GoObject**s are highly customizable. Much of this customization is achieved by setting properties, such as by setting **GoView.AllowDelete** to false. Additional customization can be accomplished by registering event handlers, such as for **GoView.ObjectGotSelection** or **GoView.ObjectSingleClicked**. More powerful customization can be achieved through the subclassing of the **GoView**, **GoTool**, **GoDocument** and **GoObject** classes and overriding their methods.

Events

Just as there are two kinds of ways to cause changes, interactive and programmatic, there are two kinds of events that Go provides, interactive and programmatic.

Interactive events are caused by a user's gestures with the mouse or the keyboard. This causes **GoView** to raise events. Examples include **GoView.ObjectDoubleClicked** and **GoView.SelectionDeleted**. These events are more suited for the purposes of an interactive diagram control than the standard **Control** events such as **Control.MouseUp**.

Programmatic events occur whenever any code anywhere causes a change, not necessarily as a result of a direct user interaction.

For changes to a **GoDocument**, its **GoLayer**s, or its **GoObject**s, the **GoDocument.Changed** event notifies all event handlers that something has changed, and it provides a description of the change in a **GoChangedEventArgs** object. As a programming convenience, **GoView** passes on **GoDocument.Changed** events as its own **GoView.DocumentChanged** event.

For changes to a **GoView**'s properties, the **GoView.PropertyChanged** event notifies its event handlers that a **GoView** property has changed. For changes to layers and objects that are part of the view (and not of the view's document), there is no defined event—only the **GoView.RaiseChanged** method is called, because needing to handle such cases is rare.

Graphical Objects

All graphical object classes in Go inherit from **GoObject**, which in turn inherits from **System.Object**. Here are some samples:
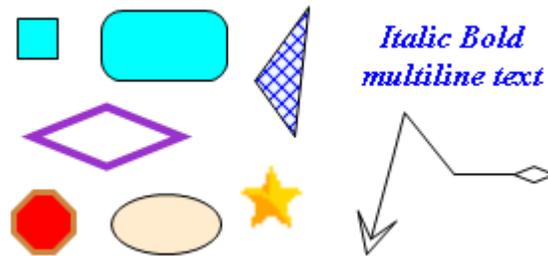
GoObject defines the basics of a graphical object: a bounding rectangle (the **Bounds** property) and some common attribute properties: **Visible**, **Printable**, **Selectable**, **Movable**, **Copyable**, **Resizable**, **Reshapable**, **Deletable**, **Editable**, **AutoRescales**, **ResizesRealtime**, and **Shadowed**.

The simplest way to think about **GoObject** is that it is a rectangular area that knows how to draw itself into a view. **GoObject** defines a virtual **Paint** method that defines the appearance of that object. Thus the full power of the

System.Drawing namespace is available for drawing your custom objects, in those rare cases where the predefined subclasses of **GoObject** do not meet your needs.

GoObject also handles certain view events and supports change notification and undo/redo. **GoObject** provides numerous methods so that custom derived objects can provide exactly the desired look and feel. More information is provided in the following chapters.

There are three kinds of primitive **GoObject**s:

- Shapes, such as rectangles, ellipses, diamonds, and strokes. Each **GoShape** instance can have a **Pen** for drawing the outline of the shape and a **Brush** for painting the inside of the shape.

- Text, in various fonts, sizes and colors. **GoText** objects also support multiple lines and wrapping. With Windows Forms, the user can perform in-place editing using several kinds of controls.

- Images, for various kinds of images such as bitmaps, JPEGs and GIFs. **GoImage** objects can get their image information from files, resource managers, or image lists.

Selection

The **GoSelection** class is used by a view to maintain a separate list of the objects selected. Each view has its own selection. In addition, the selection class notifies objects of gaining and losing selection events, and has support (in conjunction with **GoObject**) for the appearance of a selected object.
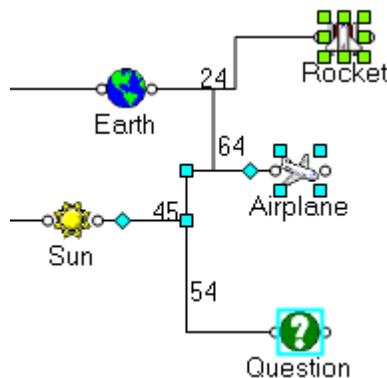
Objects can define their own selection appearance or use the default provided by the **GoObject** and **GoSelection** classes. Normally **GoSelection** uses a class called **GoHandle**, to make selection handles appear on the screen. However, you can use other objects, if they implement the **IGoHandle** interface.

GoView has some useful methods for manipulating the selection: adding objects to the selection and moving or copying or deleting the selected objects.

Collections

Go provides two principal kinds of collections of graphical objects: groups and layers. Groups provide a way of making a single "object" out of other objects. Layers are a way of viewing multiple collections of objects in a document.

GoGroup inherits from **GoObject** and implements **IGoCollection**, a collection of. **GoObject**s. Since **GoGroup** itself inherits from **GoObject**, groups can contain other groups, to any depth. An object that does not have a parent group is called a "top-level" object. The objects in a group are often called its children.

IGoCollection, and thus **GoGroup**, includes methods or properties such as **Add**, **Remove**, **Contains**, **Clear**, **IsEmpty**, **ArrayCopy**, **Count**, **GetEnumerator**, and **Backwards** so that you can manipulate the contents of the group. You can use the **foreach** construct to perform the iteration. Remember that when you are iterating over the objects in a collection you cannot modify the collection. This is true for all .NET collections.

GoGroup also provides default implementations of several **GoObject** methods such as **Paint**, **Pick**, **ComputeBounds**, and **OnBoundsChanged**. These default implementations typically iterate over all of the items in the collection, calling the appropriate methods on each object.

Since **GoGroup** is a **GoObject**, each group has a **Bounds** property. The bounds calculated by **ComputeBounds** are just the union of all of the child objects, whose coordinates are independent of the group. Moving a group will normally move all of the children; resizing a group will normally resize all of the children proportionately.

Removing a group from a document effectively causes the group's children to disappear also.

GoLayer is a collection of top-level **GoObject**s held by a **GoDocument**. Layers are just a way to organize the collection of objects owned by a document. Each document starts off with one layer. You can add and remove layers from a document. You can also change the order of the layers in a document, thereby making potentially many objects all over the document appear in front of or behind other collections of objects. Furthermore you can affect the visibility of all of those objects in a layer all at once. Unlike **GoGroup**, **GoLayer** does not extend **GoObject**, so one cannot have layers within layers. But **GoLayer** does implement **IGoCollection**, so you can use those methods and properties for manipulating the collection of objects in a layer.

Layers also support limiting user actions on the objects in the layer. Such properties include **AllowSelect**, **AllowMove**, and **AllowDelete**. For example, you can organize a document so that one layer contains all of the objects that you

do not want the user to delete; this layer would have its **AllowDelete** property set to false.

There is also a standard implementation of the **IGoCollection** interface: the **GoCollection** class. You may find this useful when you need a collection of **GoObject**s but don't want to use **GoSelection**, **GoGroup**, or **GoLayer**.

Diagrams

One of the principal uses of Go is to make it easy to build applications where users can see and manipulate diagrams (a.k.a graphs) of nodes (a.k.a. vertices) connected by links (a.k.a. arcs or edges). Go provides this functionality with the **GoNode**, **GoPort** and **GoLink** classes. Nodes are groups containing one or more ports. Links are strokes that connect two ports. Most of the predefined classes that you will use to represent your graphs or networks are subclasses of these classes.

The following picture identifies the parts of two nodes connected by a link at two ports.
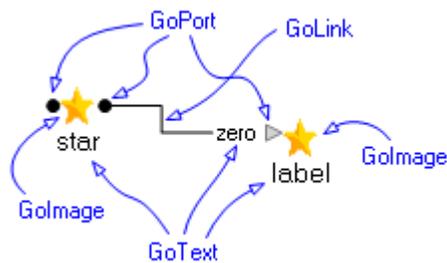
Although the **GoNode**, **GoPort**, and **GoLink** classes are all subclasses of **GoObject**, the basic aspects of being a node, a port, or a link are actually embodied by the **IGoNode**, **IGoPort**, and **IGoLink** interfaces.

IGoLink provides properties for getting the **IGoPort**s at either end of the link.

IGoPort provides access to the **IGoLink**s that are connected to the port. The members allow access to links (and therefore implicitly to the nodes that contain the ports of those links) that are coming into the port or going out of the port, under the assumption that links are directional. Or you can deal with the whole collection of links in either direction at the port. **IGoPort** also provides members for determining if the user can draw a link from one port to another port.

IGoNode provides access to the **IGoPort**s that the node contains, as well as to the collections of links or nodes that those ports are directly connected to.

All three of these interfaces provide access to the **GoObject** that represents the abstract node, port, or link. And all three interfaces (and **GoDocument** too) provide two properties that allow you to associate a custom integer value and a custom object with each node, port, or link, without having to subclass.
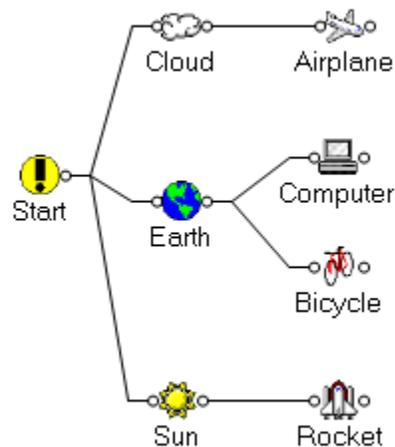
A frequent feature of nodes is that they have a distinguished or primary text label. The **IGoLabeledNode** interface provides access to both the string value and the **GoText** label. **GoNode** implements **IGoLabeledNode** too, so methods such as **GoDocument.FindNode** can search for a node that matches a string.

The sample classes provide some pre-built implementations of useful nodes, in addition to the ones that are included in Go. You can modify them if you need to customize their appearance or behavior.

The sample apps provide some pre-built implementations of graphical browsers and editors, using the views and nodes that Go supplies. They show you how to create diagrams and how to load and store them from files.

A Minimal Application

A very basic use of Go is provided in the samples directory, **MinimalApp**.

This minimal application just puts up two **BasicNode**s of different colors. The user can link them together, select nodes and/or links, move them around, copy them, or delete them. Go provides all of this functionality automatically—the code just needs to create the nodes and add them to the view's document.
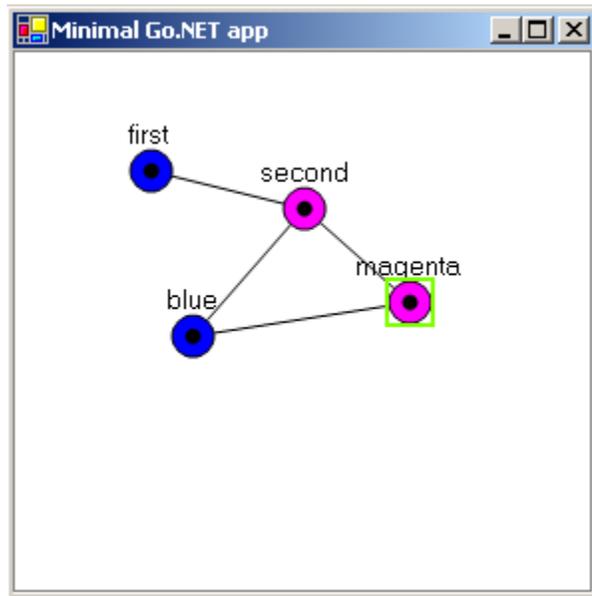
This shows how the **MinimalApp** appears after selecting the two initial nodes, drag-copying them, moving them, creating links between some of them, and then renaming a blue one to "blue" and a magenta one to "magenta".

Windows Forms VB.NET:

```vb
Imports Northwoods.Go

Public Class MinimalApp
  Inherits Form
  ' constructor
  Public Sub New()
    MyBase.New()
    Me.Text = "Minimal GoDiagram app"
    ' create a Go view (a RichControl) and add to the form
    Dim myView As GoView = New GoView()
    myView.Dock = DockStyle.Fill
    Me.Controls.Add(myView)

    ' create two nodes for fun...
    Dim node1 As GoBasicNode = New GoBasicNode()
    ' specify position, label and color
    node1.Location = New PointF(100, 100)
    node1.Text = "first"
    node1.Editable = True  ' first node is editable with F2 only
    node1.Shape.BrushColor = Color.Blue
    ' add to the document, not to the view
    myView.Document.Add(node1)

    Dim node2 As GoBasicNode = New GoBasicNode()
    node2.Location = New PointF(200, 100)
    node2.Text = "second"
    node2.Label.Editable = True ' second node editable by clicking only
    node2.Shape.BrushColor = Color.Magenta
    myView.Document.Add(node2)
  End Sub

  Shared Sub Main()
    Application.Run(New MinimalApp())
  End Sub
End Class
```

Windows Forms C#:

```csharp
using Northwoods.Go;

public class MinimalApp : Form {
  // constructor
  public MinimalApp() {
    this.Text = "Minimal GoDiagram app";
    // create a Go view (a Control) and add to the form
    GoView myView = new GoView();
    myView.Dock = DockStyle.Fill;
    this.Controls.Add(myView);

    // create two nodes for fun...
    GoBasicNode node1 = new GoBasicNode();
    // specify position, label and color
    node1.Location = new PointF(100, 100);
    node1.Text = "first";
    node1.Editable = true;  // first node is editable with F2 only
    node1.Shape.BrushColor = Color.Blue;
    // add to the document, not to the view
    myView.Document.Add(node1);

    GoBasicNode node2 = new GoBasicNode();
    node2.Location = new PointF(200, 100);
    node2.Text = "second";
    node2.Label.Editable = true; // node editable by clicking only
    node2.Shape.BrushColor = Color.Magenta;
    myView.Document.Add(node2);
  }

  [STAThread]
  public static void Main(string[] args) {
    Application.Run(new MinimalApp());
  }
}
```

GoDiagram User Guide

Compact Framework Windows Forms C#:

```csharp
using Northwoods.Go;

public class MinimalApp : Form {
  // constructor
  public MinimalApp() {
    this.Text = "Minimal GoDiagram app";
    // create a Go view (a Control) and add to the form
    GoView myView = new GoView();
    myView.Bounds = this.ClientRectangle;
    this.Controls.Add(myView);

    // create two nodes for fun...
    GoBasicNode node1 = new GoBasicNode();
    // specify position, label and color
    node1.Location = new PointF(100, 100);
    node1.Text = "first";
    node1.Label.Editable = true;
    node1.Shape.BrushColor = Color.Blue;
    // add to the document, not to the view
    myView.Document.Add(node1);

    GoBasicNode node2 = new GoBasicNode();
    node2.Location = new PointF(200, 100);
    node2.Text = "second";
    node2.Label.Editable = true;
    node2.Shape.BrushColor = Color.Magenta;
    myView.Document.Add(node2);
  }

  public static void Main(string[] args) {
    Application.Run(new MinimalApp());
  }
}
```

ASP.NET Web Forms:

WebForm1.aspx:

```
<%@ Page language="c#" Codebehind="WebForm1.aspx.cs"
    AutoEventWireup="false" Inherits="MinimalApp.WebForm1" %>
<%@ Register TagPrefix="GoWeb" Namespace="Northwoods.GoWeb"
    Assembly="Northwoods.GoWeb" %>
. . .
<GoWeb:GoView id="MyView" Runat="Server" TabIndex="1"
              Height="300" Width="400"></GoWeb:GoView>
. . .
<button id="CopyButton" onclick="goAction('copy','MyView')"
        type="button">Copy</button>
. . .
```

WebForm1.aspx.cs:

```
public class WebForm1 : System.Web.UI.Page {
  protected Northwoods.GoWeb.GoView MyView;

  private void Page_Load(object sender, System.EventArgs e) {
    MyView.SessionStarted += new EventHandler(InitializeDocument);
  }

  private void InitializeDocument(Object sender, EventArgs evt) {
    // create two nodes for fun...
    GoBasicNode node1 = new GoBasicNode();
    // specify position, label and color
    node1.Location = new PointF(100, 100);
    node1.Text = "first";
    node1.Shape.BrushColor = Color.Blue;
    // add to the document, not to the view
    MyView.Document.Add(node1);

    GoBasicNode node2 = new GoBasicNode();
    node2.Location = new PointF(200, 100);
    node2.Text = "second";
    node2.Shape.BrushColor = Color.Magenta;
    MyView.Document.Add(node2);

    // support undo/redo
    MyView.Document.UndoManager = new GoUndoManager();
  }
  . . .
}
```

3. BUILDING APPLICATIONS

This chapter describes some of the typical tasks involved in building an application using Go.

Choosing a Model

Before you start implementing your own application, you should have a clear model in your mind for how your "real world" information can be organized into a diagram that could be drawn with nodes and links. Often one of the sample applications that Go provides will look like what you want.

Select Node and Link Types

Besides the basic graphical objects that Go provides, there are a number of parts that help you build diagrams or networks of objects. In particular, there are several predefined subclasses of **GoNode** that are used by the various sample applications. Chapter 6. discusses these node classes in more detail.

Depending on the kind of application you are building, one of these existing node classes is likely to be close to what you need, even if the appearance isn't right. Choose from the supplied node classes by deciding if it needs an image and how many ports it should have. Initially you may wish to use them as-is, and concentrate on having the diagram reflect the structure of your "real world" model and have it handle user edits. Later you can customize and elaborate the state, appearance, and behavior of the parts and the diagram as a whole.

Define Property Editors

Another common task is implementing property-editing forms for each kind of node or link. Typically these forms are displayed in response to the F4 key for the primary selection or a double click on a node or link.

Depending on the nature of the properties you need to display and allow the user to edit, you can either implement custom dialogs, or you can use **PropertyGrid**s. Both approaches are used in the ProtoApp sample in GoDiagram Win.

Customize Node and Link Appearances

Eventually you will probably want to add graphics that are specific to the real object the node represents. For example, if the node is a shop floor manufacturing machine, there might be a "Stopped" state that might change the appearance of the node so the operator could tell at a glance.

But not all of the interesting information can or should be shown as **GoObject**s. For example, additional status and a lot of controls for that shop floor manufacturing machine probably belong in a dialog.

Programmatically Creating a Diagram with Nodes and Links

Users can easily build and modify diagrams if you give them the ability to insert nodes. But often you will want to build a diagram programmatically—where your code will create and insert nodes and find and link ports. Your code will also need to update the persistent data storage with changes that the user has made.

Here are the steps your code will need to take:

1. Allocate a new instance of a node class

2. Initialize the new node by setting its properties and calling appropriate methods on it. You'll want to set its **Position** at this time.

3. Add the new node to the document.

4. Repeat to create other nodes.

5. For each link, first find the proper port on the source node and the proper port on the destination port. This typically involves finding the appropriate source and destinations nodes first, and then identifying the desired output and input ports on the respective nodes.

6. Allocate a new instance of a link class

7. Set its **FromPort** and **ToPort** properties, and any other desired properties, and call any other initializing methods.

8. Add the new link to the document.

9. Repeat to create other links.

C#:
```csharp
GoBasicNode node1 = new GoBasicNode();
node1.Location = new PointF(100, 100);
node1.Text = "first";
node1.Shape.BrushColor = Color.Blue;
goView1.Document.Add(node1);

GoBasicNode node2 = new GoBasicNode();
node2.Location = new PointF(200, 100);
```

```
node2.Text = "second";
node2.Shape.BrushColor = Color.Magenta;
goView1.Document.Add(node2);

GoLink link = new GoLink();
link.ToArrow = true;
link.PenColor = Color.Orange;
link.FromPort = node1.Port;
link.ToPort = node2.Port;
goView1.Document.Add(link);
```

VB.NET:
```
Dim node1 As GoBasicNode = New GoBasicNode()
node1.Location = New PointF(100, 100)
node1.Text = "first"
node1.Shape.BrushColor = Color.Blue
goView1.Document.Add(node1)

Dim node2 As GoBasicNode = New GoBasicNode()
node2.Location = New PointF(200, 100)
node2.Text = "second"
node2.Shape.BrushColor = Color.Magenta
goView1.Document.Add(node2)

Dim link As GoLink = New GoLink()
link.ToArrow = True
link.PenColor = Color.Orange
link.FromPort = node1.Port
link.ToPort = node2.Port
goView1.Document.Add(link)
```

This is the basic idea whenever you need to build a diagram programmatically—whether the real information is contained in a database or file or comes from some other source.

As you create nodes, you will want to make sure that each node has the key information it needs to uniquely identify the ultimate information source. For example, each node you create may want to have a unique label that you can use to look up the right row in a database table. Or you could use the node's **UserFlags** property or **UserObject** property for holding the key.

If you have more than one port on a node, you may need to add similar identifying information to each port also.

Handling Events

GoView has many properties that control how it appears and how it behaves when the user tries to interact with it. But the bulk of the customization is accomplished by defining event handlers. Although Chapter 5. discusses

GoView properties and event handlers in detail, we can give a few examples here.

If you want to do something when the user double clicks on a node, you might add an **ObjectDoubleClicked** event handler to the view.

VB.NET:

```
Private WithEvents goView1 As GoView = new GoView()

Protected Sub goView1_ObjectDoubleClicked(ByVal sender As Object,
    ByVal e As GoObjectEventArgs) Handles goView1.ObjectDoubleClicked
  If Not PointToSelectCheckBox.Checked Then
    If TypeOf e.GoObject.TopLevelObject Is GoIconicNode Then
      Dim n As GoIconicNode = CType(e.GoObject.TopLevelObject,
                                    GoIconicNode)
      MessageBox.Show("Action invoked on " + n.Text)
    End If
  End If
End Sub
```

C#:

```
    . . .  // other Form initialization
    this.goView1 = new GoView();
    this.goView1.ObjectDoubleClicked +=
        new GoObjectEventHandler(this.goView1_ObjectDoubleClicked);
    . . .

  private void goView1_ObjectDoubleClicked(object sender,
                                           GoObjectEventArgs e) {
    GoObject obj = e.GoObject;
    // get the top-level object for the object that got
    // double-clicked and see if it is a BasicLayoutNode
    BasicLayoutNode n = obj.TopLevelObject as BasicLayoutNode;
    if (n != null)
      n.ChangeColor(); // if found, change its color
  }
```

GoObjectEventArgs includes additional information about the input event besides the **GoObject** that it happened at—**GoObjectEventArgs.DocPoint** describes where the mouse event occurred in the document. The state of the mouse buttons and other information are available as well.

Other **GoView** events do not involve any particular mouse or keyboard input. For example, when the current selection is deleted, there are events that occur just before and just after. The **SelectionDeleting** event is cancelable—setting the **CancelEventArgs.Cancel** property to true avoids removing the selected objects from the document. The **SelectionDeleted** event occurs after the objects have been removed from the document.

VB.NET:

```
Protected Sub goView1_SelectionDeleting(ByVal sender As Object,
       ByVal evt As CancelEventArgs) Handles goView1.SelectionDeleting
   If MessageBox.Show("Delete " + goView1.Selection.Count + " objects?",
                      "About to delete selection",
                      MessageBoxButtons.YesNo) = DialogResult.No Then
       evt.Cancel = True
   End If
End Sub

Protected Sub goView1_SelectionDeleted(ByVal sender As Object,
       ByVal evt As EventArgs) Handles goView1.SelectionDeleted
   MessageBox.Show(goView1.Document.Count + " objects left")
End Sub
```

C#:
```
protected void goView1_SelectionDeleting(object sender,
                                         CancelEventArgs evt) {
   if (MessageBox.Show("Delete " + goView1.Selection.Count+ " objects?",
                       "About to delete selection",
                       MessageBoxButtons.YesNo) == DialogResult.No) {
     evt.Cancel = true;
   }
}

private void goView1_SelectionDeleted(object sender, EventArgs e) {
   MessageBox.Show(goView1.Document.Count + " objects left");
}
```

Following the .NET convention for naming and defining events, you are encouraged to override the protected **GoView.On…** method for an event instead of adding an event handler, if you have defined your own subclass of **GoView**. The behavior is the same but more efficient; furthermore the code is then naturally part of the view rather than jumbled together with other code in the form. Remember to call the base method to make sure all event handlers get called.

VB.NET:
```
Protected Overrides Sub OnObjectGotSelection(ByVal evt
                                             As GoSelectionEventArgs)
   MyBase.OnObjectGotSelection(evt)
   If Not myPrimarySelection Is Me.Selection.Primary Then
     myPrimarySelection = Me.Selection.Primary
     ' update the toolbar to match the selection
     MainForm.App.EnableToolBarEditButtons(Me)
   End If
End Sub
```

C#:
```
protected override void OnObjectGotSelection(GoSelectionEventArgs evt)
{
```

```
    base.OnObjectGotSelection(evt);
    if (myPrimarySelection != this.Selection.Primary) {
      myPrimarySelection = this.Selection.Primary;
      // update the toolbar to match the selection
      MainForm.App.EnableToolBarEditButtons(this);
    }
}
```

However there are also events that occur not as the result of any direct user interaction, but due to changes to a document or to objects in a document. These programmatic events are **GoDocument.Changed** events. To make it easier to define document change event handlers, these **GoDocument** events are passed through by **GoView** as **GoView.DocumentChanged** events.

For example, the following event handler notices when any code inserts a node into the document and updates a **ComboBox** correspondingly.

VB.NET:
```
  Private Sub goView1_DocumentChanged(ByVal sender As Object,
        ByVal e As GoChangedEventArgs) Handles goView1.DocumentChanged
    Select Case e.Hint
      Case GoLayer.InsertedObject
        ' added a node to the document--gotta add it to
        ' the combobox's list of nodes
        If TypeOf e.Object Is IGoNode Then
          Dim n As IGoNode = CType(e.Object, IGoNode)
          If Not n.UserObject Is Nothing Then
            Me.NodeCombo.Items.Add(n.UserObject)
          End If
        End If
      . . .  ' other kinds of cases
    End Select
  End Sub
```

C#:
```
  private void goView1_DocumentChanged(object sender,
                                       GoChangedEventArgs e) {
    switch (e.Hint) {
    case GoLayer.InsertedObject: {
        // added a node to the document--gotta add it to
        // the combobox's list of nodes
        IGoNode n = e.Object as IGoNode;
        if (n != null && n.UserObject != null) {
          this.NodeCombo.Items.Add(n.UserObject);
        }
        break;
      }
      . . .  // other kinds of changes
    }
  }
```

You can get the exact same results more efficiently by defining your own **GoDocument** subclass and overriding **GoDocument.OnChanged**.

Traversing a Diagram

This section includes some examples of how to traverse a diagram. A diagram is normally implemented with **GoObject** classes such as **GoTextNode** and **GoPort** and **GoLabeledLink**. However, a more abstract way of dealing with nodes and links is provided by several interfaces: **IGoNode**, **IGoPort**, **IGoLink**. You can define general graph traversing algorithms without having to worry about the exact classes used to implement the parts of the graph.

IGoNode

IGoNode represents an abstract node, containing one or more **IGoPort**s. The **IGoNode.Ports** property provides an enumerable so that you can iterate over all of the ports of a node.

To make it more convenient to get to all of the links connected to a node, regardless of the port that they are connected to, there are some properties that provide enumerators over links. The **IGoNode.Links** property lets you iterate over all of the links connected at a node; the **IGoNode.SourceLinks** and **IGoNode.DestinationLinks** properties just enumerate over the links coming into or going out of a node.

Finally to make it more convenient to get to all of the nodes that are connected to a node, the **IGoNode.Nodes** property provides an enumerator for iterating over all of the nodes that have any direct connection to any port of a node. Again, if you only want to consider those nodes that are at one particular end of directed links, you can use the **IGoNode.Sources** and **IGoNode.Destinations** properties.

IGoPort

IGoPort represents a part of a node that is like a socket for holding the ends of some **IGoLink** connections. The **IGoPort.Links** property gets an enumerable to iterate over all of the links connected at an abstract port. Because links usually considered to be directed, the **IGoPort.SourceLinks** and **IGoPort.DestinationLinks** properties let you iterate over only those links coming into or going out of a port.

You can also manipulate an abstract port by using the **IGoPort.AddSourceLink**, **IGoPort.AddDestinationLink**, **IGoPort.RemoveLink**, and **IGoPort.ContainsLink** methods.

IGoPort.Node is a property to allow you to navigate from a port to its containing node.

IGoPort also defines three predicates that are useful in deciding whether it is valid to create a link between two ports. The **IGoPort.IsValidLink** predicate is

the primary method; the **IGoPort.CanLinkFrom** and **IGoPort.CanLinkTo** predicates are typically called by implementations of **IsValidLink** to see if there are any port-specific reasons why a link should not be permitted, in addition to the decisions that **IsValidLink** should make considering both ports.

IGoLink

IGoLink represents a connection between two **IGoPort**s. It defines two properties, the **IGoLink.FromPort** and **IGoLink.ToPort**, as the principal properties of any link. For convenience, the **IGoLink.FromNode** and **IGoLink.ToNode** properties are also defined, to return the **IGoPort.Node** property of the corresponding port.

The **IGoLink.FromPort** and **IGoLink.ToPort** properties are defined to be settable. The **IGoLink.Unlink** method is defined to disconnect the link from both ports and remove the link from any container.

To make it easier to traverse graphs by following links in either direction, **IGoLink** defines two methods for getting from one end of the link to the other, without assuming which end you have to begin with: **IGoLink.GetOtherPort** and **IGoLink.GetOtherNode**.

IGoGraphPart

IGoGraphPart is the base interface for the **IGoNode**, **IGoPort**, and **IGoLink** interfaces. It defines the **GoObject** property for getting an object that may be part of a **GoDocument**. It also defines the **UserFlags** and **UserObject** properties that may be implemented by classes to hold an integer and an object associated with the part of the diagram.

There are standard implementations of these interfaces that are also **GoObject**s: **GoNode** implements **IGoNode**, **GoPort** implements **IGoPort**, and **GoLink** and **GoLabeledLink** implement **IGoLink**. Since any instance of **GoNode**, **GoPort**, **GoLink**, or **GoLabeledLink** are also instances of **GoObject**, the **IGoGraphPart.GoObject** property just returns itself. All of these classes also provide storage for the **UserFlags** and **UserObject** properties that you can set.

Simple Traversal Examples

Iterating over all of the nodes in a document:

VB.NET:
```
For Each obj In aDocument
  If TypeOf obj Is IGoNode Then
    Dim n As IGoNode = CType(obj, IGoNode)
    ' do something with the IGoNode n, typically a GoNode
  End If
Next obj
```

C#:
```csharp
foreach (GoObject obj in aDocument) {
  IGoNode n = obj as IGoNode;
  if (n != null) {
    // do something with the IGoNode n, typically a GoNode
  }
}
```

Alternatively, if you know the node class you are looking for:

VB.NET:
```vbnet
For Each obj In aDocument
  If TypeOf obj Is GraphNode Then
    Dim n As GraphNode = CType(obj, GraphNode)
    ' do something with the GraphNode n
  End If
Next obj
```

C#:
```csharp
foreach (GoObject obj in aDocument) {
  GraphNode n = obj as GraphNode;
  if (n != null) {
    // do something with the GraphNode n
  }
}
```

Iterating over all of the links in a document:

VB.NET:
```vbnet
For Each obj In aDocument
  If TypeOf obj Is IGoLink Then
    Dim link As IGoLink = CType(obj, IGoLink)
    ' do something with the IGoLink link,
    ' which is typically a GoLink or a GoLabeledLink
  End If
Next obj
```

C#:
```csharp
foreach (GoObject obj in aDocument) {
  IGoLink link = obj as IGoLink;
  if (link != null) {
    // do something with the IGoLink link,
    // which is typically a GoLink or a GoLabeledLink
  }
}
```

Alternatively, if you know the link class you are looking for:

VB.NET:
```vbnet
For Each obj In aDocument
  If TypeOf obj Is GraphLink Then
    Dim link As GraphLink = CType(obj, GraphLink)
    ' do something with the GraphLink link,
  End If
Next obj
```

C#:
```csharp
foreach (GoObject obj in aDocument) {
  GraphLink link = obj as GraphLink;
  if (link != null) {
    // do something with the GraphLink link,
  }
}
```

Selecting all the nodes directly connected to a node labeled "Rome":

VB.NET:
```vbnet
Dim obj As GoObject = aView.Document.FindNode("Rome")
If TypeOf obj Is CityNode Then
  Dim rome As CityNode = CType(obj, CityNode)
  For Each node In rome.Nodes
    If TypeOf node.GoObject Is CityNode Then
      aView.Selection.Add(node.GoObject);
    End If
  Next node
End If
```

C#:
```csharp
CityNode rome = aView.Document.FindNode("Rome") as CityNode;
if (rome != null) {
  foreach (IGoNode n in rome.Nodes) {
    if (n.GoObject is CityNode)
      aView.Selection.Add(n.GoObject);
  }
}
```

Finding all direct flights from one city to another:

VB.NET:
```vbnet
Dim orig As CityNode = CType(aDocument.FindNode("Madrid"), CityNode)
Dim dest As CityNode = CType(aDocument.FindNode("Berlin"), CityNode)
Dim results As GoCollection = new GoCollection()
For Each l in orig.DestinationLinks
```

```
  If l.ToNode Is dest Then
    results.Add(l.GoObject)
  End If
Next l
```

C#:

```
CityNode orig = aDocument.FindNode("Madrid") as CityNode;
CityNode dest = aDocument.FindNode("Berlin") as CityNode;
GoCollection results = new GoCollection();
foreach (IGoLink l in orig.DestinationLinks) {
  if (l.ToNode == dest)
    results.Add(l.GoObject);
}
```

More Complex Traversals

The Demo1 sample includes code that finds and highlights the longest path(s) of nodes coming out of a selected node. The code, in the **GraphView** class, demonstrates one technique for finding the distance from a root node for all of the nodes that are reachable from that root node. It keeps the distances in a **Hashtable**, and uses an **ArrayList** to remember the current path as it is traversing the diagram in order to avoid cycles.

Once it finds the nodes that are the furthest away from the root node, it walks backwards from those nodes, through the source links, highlighting the actual **GoLink** or **GoLabeledLink** as it goes.

Of course, this is only one implementation of one variation of a path-finding algorithm. Many other kinds of path-finding tasks are needed for various applications, and there are many other kinds of graph algorithms that you may find useful. If you don't already know what you need to do, there are many books available that discuss these issues.

There are some static/shared methods on **GoDocument** that are used to implement the **GoDocument.ValidCycle** property: **MakesDirectedCycle**, **MakesDirectedCycleFast**, **MakesUndirectedCycle**. These methods are called by **GoPort.IsValidLink** to see if there are any kinds of cycles that might be introduced into the graph.

Supporting Save and Load

Go does not have a standard file format that you have to use. For your diagrams, you will need to implement code to save to and load from whatever data store is appropriate for your application the information that the diagram represents. A number of samples implement persistence using a simple custom XML format. You can read more about GoXml and the **GoXmlBindingTransformer** class in a later chapter in this User Guide. The DataSetDemo sample demonstrates two-way updating with a DataSet.

Node and link specific data can initially be stored in the **UserObject** property for nodes and links. (This property is just like the **Tag** property for **TreeNode**s, but there is also a **UserFlags** property for storing an integer efficiently.) Later you may want to create subclasses that have fields holding this information.

When it is time to store the diagram, you can traverse the diagram looking at all of the nodes. For each node that has key identifier information, use the key to find the corresponding record, and update the record appropriately. Each node that does not have this key information you will know to be a new node, and you can insert a new record.

Determining which records to delete can be achieved in several ways. For example, you can query the database to get all of the records. You can delete each record for which no node exists that has the corresponding key information.

An alternative method for determining which records to delete is to keep track of which nodes are deleted. Add a document **Changed** event handler (or equivalently, override **GoDocument.OnChanged** or override **GoView.OnDocumentChanged**) to detect events with a **GoLayer.RemovedObject** hint. If the object is the right kind of node class, remember either a reference to the node or the key information, in a list of deleted nodes or deleted keys. Then the diagram storage process just needs to run through the list and delete the corresponding records, followed by clearing out the list.

To simplify the generation of unique IDs for nodes and ports and links, **GoDocument** has a property that automatically makes sure that each node, port, or link that is added to the document has a unique **PartID**. Just set the **GoDocument.MaintainsPartID** to true. All objects that implement the **IGoIdentifiablePart** interface provide a **PartID** property; this is set by **GoDocument** as objects are added to the document.

When you need to refer to objects, such as to the ports of a link that you are storing, you can just pass the **PartID**. Upon loading, you can find the **IGoIdentifiablePart** in the document with that ID by calling **GoDocument.FindPart**. Remember to save the **LastPartID** in your document too, to avoid any possible duplicate PartIDs.

Of course, you can implement your own mechanism for keeping track of identities, instead of using **PartID**. Typically you will have one or more hash tables used to map key values to nodes and perhaps ports if there might be more than one port in a node.

4. DOCUMENTS AND GOOBJECTS

GoDocument

GoDocument represents a group of **GoObject**s that can be displayed by a **GoView**. **GoDocument** represents the model in the model-view-controller architecture; **GoView** and **GoTool** play the role of the view and controller.

A document is a collection of objects, organized into layers. The layers are ordered. The layers are drawn in sequential order, so objects in layers toward the beginning of the list of layers are drawn first and thus appear "behind" objects that are in later layers. You can add, remove, and iterate over the document's objects by using the document's implementation of the **IGoCollection** interface

In addition to all of the objects held by the document, the document has its own notion of the background color, called the paper color. This is independent of and takes precedence over the **GoView** background color (i.e. **Control.BackColor**). By default the document has no paper color (**Color.Empty**), so the view's **BackColor** will appear. But when the document has a non-empty paper color, all views will use that paper color as the background.

For your convenience, each document has a **Name** property that you may use for identification purposes. Initially it is an empty string. The **Name** property is used as the document name when printing.

GoDocument also supports undo and redo by cooperating with a **GoUndoManager** that observes and records changes to the document.

Layers

A layer is just a collection of **GoObject**s. Although you normally think of a document as owning the objects in it, actually a document directly owns only an instance of **GoLayerCollection**, which is a collection of **GoLayer**s. Each layer in turn owns all of the objects in its collection. Each **GoObject** can be part of at most one **GoLayer**.

When you have created an instance of a **GoObject**, you'll want to add it to a document by calling the **GoDocument.Add** method. This actually adds the object to a particular layer in the document, the **GoDocument.DefaultLayer**,

unless it implements **IGoLink**, in which case it adds the link to the **GoDocument.LinksLayer**. You may wish to ensure an object appears in front of or behind other objects. If so, you should make sure the appropriate layer exists and then **Add** the object to that layer.

You can use the **GoLayer.AddCollection** method for adding a collection of objects to a layer. This method can even move objects from within **GoGroup**s to be top-level objects, without disconnecting any links as would normally happen if objects are first **Remove**d and then **Add**ed.

You can affect the ordering of objects within a layer by calling the **GoLayer.MoveBefore** or **MoveAfter** methods. If you want to inquire about the relative ordering of some objects, you can call the **GoLayerCollection.SortByZOrder** method to sort a given array of **GoObject**s. (This method does not modify the ordering or layers for any objects.)

Initially a document has one layer that is used for holding all objects added to the document. Documents also support the notion of a layer for holding all links that the user creates. This **GoDocument.LinksLayer** property is the layer to which the linking tool adds newly created links. By default, since there is initially only one layer in a document, this layer will be the same as for all other objects in the document. When there is only one layer, you cannot be sure whether links will appear in front of or behind any nodes. But if you explicitly create a new layer and assign the **LinksLayer** property to this new layer, you can control this appearance. For example,

```
doc.LinksLayer = doc.Layers.CreateNewLayerAfter(doc.Layers.Default)
```

will ensure that all user drawn links will appear in front of all nodes inserted in the default layer. You should always add programmatically created links into the document's links layer, using a call such as:

```
doc.LinksLayer.Add(aNewLink)
```

Enumerating the objects in a layer can be done either forwards or backwards, because **IGoCollection** supports both regular (forwards) and backwards iteration. Both directions are needed so that painting can be done in the opposite direction from picking. This ensures that the user will always pick the front-most object that can be seen at a particular spot.

For example, the following code finds a graphical object in the document's default layer that is the furthest left (i.e. has the smallest X coordinate). It ignores objects in other layers.

VB.NET:
```
Dim leftx As Single = 1.0E+20   F
Dim leftmost As GoObject = Nothing
Dim obj As GoObject
For Each obj In doc.DefaultLayer
  If obj.Left < leftx Then
    leftx = obj.Left
```

```
      leftmost = obj
    End If
  Next
```

C#:

```
float leftx = 1.0e20f;
GoObject leftmost = null;
foreach (GoObject obj in doc.DefaultLayer) {
  if (obj.Left < leftx) {
    leftx = obj.Left;
    leftmost = obj;
  }
}
```

You can programmatically find the front-most object at a particular point by calling **GoDocument.PickObject**. To find some or all of the objects at a point, even if hidden behind some objects, you can use the **GoDocument.PickObjects** method. (**GoLayer** and **GoView** also have similar methods.)

In order to support the **GoLink.AvoidsNodes** property, each **GoDocument** can keep track of where "avoidable" nodes are. The **GoDocument.IsAvoidable** predicate and **GetAvoidableRectangle** method determine which document objects to consider avoidable and what their effective bounds are. By default all objects implementing **IGoNode** are considered avoidable.

You can programmatically ask if a particular rectangular area may have any avoidable objects in it with the **GoDocument.IsUnoccupied** predicate.

The use of this avoidance functionality can be computationally expensive, so you should only use it when it is necessary.

Layers are normally owned by documents, but some layers will be owned by views instead. Objects owned by layers owned by documents are called "document objects"; those in layers owned by views are called "view objects". **GoObject.Document** and **GoLayer.Document** are both non-null for document objects. The same holds for **GoObject.View** and **GoLayer.View** and view objects. Use **GoObject.IsInDocument** or **GoLayer.IsInDocument** to determine if something is a document object or a document layer. Note that **GoObject.IsInDocument** will be true if the object is in any layer of the document.

Layer Abilities

Layers also implement the **IGoLayerAbilities** interface, which defines the properties and methods used by Go to determine if the user may perform certain operations. These are:

- **CanSelectObjects**, **AllowSelect**

- **CanMoveObjects**, **AllowMove**

- **CanCopyObjects**, **AllowCopy**

- **CanResizeObjects**, **AllowResize**

- **CanReshapeObjects**, **AllowReshape**

- **CanDeleteObjects**, **AllowDelete**

- **CanInsertObjects**, **AllowInsert**

- **CanLinkObjects**, **AllowLink**

- **CanEditObjects**, **AllowEdit**

GoDocument and **GoView** also implement **IGoLayerAbilities**, so one can declaratively control the behavior of objects in a layer or in all layers of a document by setting the **Allow***ACT* property (for any ability *ACT*), or one can override the implementation of the **Can***ACT***Objects** method. Most of these abilities also apply to individual **GoObject**s; inserting and linking do not because they do not involve exactly one object.

As an example, **GoView.CanDeleteObjects()** will be true if **GoView.AllowDelete** is true and if the view's document's **CanDeleteObjects()** is true. The **AllowDelete** property is a browsable one, so that you can easily disable deleting objects in a particular view by editing its properties in your .NET IDE.

GoDocument.CanDeleteObjects() will be true if **GoDocument.AllowDelete** is true.

GoLayer.CanDeleteObjects() will be true if **GoLayer.AllowDelete** is true and if the layer's document's **CanDeleteObjects()** is true.

GoObject.CanDelete() will be true if **GoObject.Deletable** is true and if the object's layer's **CanDeleteObjects()** method is true.

Thus you can make an object not deletable by the user by setting a property to false at one or more of three different levels of the document object hierarchy: object, layer, and document. Furthermore you can make all objects not deletable for a particular view (but not necessarily for other views on the same document) by setting the view's **AllowDelete** property to false.

For convenience, the **SetModifiable** method allows one to set the move, resize, reshape, delete, insert, link, and edit ability properties all at once.

Layers have an identifier property that you can use to distinguish different layers. The layer that every document starts off with as the default layer has an identifier that is the **Integer** zero, but otherwise each layer initially has no identifier.

Document Coordinates and Size

The **GoObject**s held in the document have a size and position. The coordinate system used by the document is the same as the default coordinate system for

controls, i.e. positive coordinates increase rightwards and downwards and each unit corresponds to a pixel. **GoView**s have a coordinate system that may be translated and scaled from that of the document, so as to support panning and zooming.

Document coordinates use **single** (**float**) values. Thus **GoObject** sizes and offsets are held by **SizeF** structures, positions by **PointF** structures, and bounds by **RectangleF** structures. View coordinates, like all Control coordinates, use **int**eger values (and thus **Size**, **Point**, and **Rectangle** structures).

The document's size is automatically expanded to encompass all of its objects. Normally a document has all of its objects at positive coordinates (i.e., the lower right quadrant). However, if there are objects with negative coordinates, the **GoDocument.TopLeft** property will indicate the actual "origin". This property combined with the **GoDocument.Size** property gives the full extent of all of the objects in the document. It is possible to set either of these properties, but by default they will automatically get re-set as existing objects are moved or resized or as new objects are added. If you want to keep the document **Size** and **TopLeft** properties constant, regardless of where any objects are placed in the document, you can set the **GoDocument.FixedSize** property to true. If you need different behavior, you will need to override **GoDocument.UpdateDocumentBounds**, which is responsible for keeping the document size and top-left up-to-date as objects are changed or added.

The normal behavior is that the **Size** property is increased to accommodate objects that are placed beyond where they had been before. However, the **Size** property does not automatically shrink—not even when all of the objects in the document are removed and the document is empty. The **Size** property also always includes the (0, 0) point. If you want to find out how much coordinate space all of a document's objects are actually taking, use the **GoDocument.ComputeBounds** method. This method calculates the union of the bounds of all of the document objects for which **GoObject.CanView()** is true, or for which **GoObject.CanPrint()** is true when the **GoView.IsPrinting** property is true. If you want to automatically shrink the document's extent as objects are moved or removed, you will need to override **GoDocument.OnChanged** to notice when objects are removed, and override **UpdateDocumentBounds** to calculate any extent adjustments.

Events

GoDocument produces one event, **Changed**. Following the standard naming conventions, the **GoChangedEventArgs** class provides a description of any changes to a document, and the delegate **GoChangedEventHandler** takes a **GoChangedEventArgs** as a second argument.

Each **GoView** adds a **GoChangedEventHandler** delegate to its document, resulting in calls to **GoView.OnDocumentChanged**. Each view needs to notice when documents and document objects change so that it can update the visible

rendering of that document and those objects. You can register your own event handlers to notice changes to the document or its objects.

The default implementation of **GoDocument.OnChanged** invokes all of the document's event handlers. Normally, though, you will call the **GoDocument.RaiseChanged** method to take care of notifying event handlers. This method calls **OnChanged** but is more efficient and easier to call because you won't have to construct a **GoChangedEventArgs** argument. **RaiseChanged** puts its arguments into an instance of **GoChangedEventArgs** before calling **OnChanged**. You can override **OnChanged** or **RaiseChanged** if you want your document subclass to always respond to certain **Changed** events without having to register an event handler with itself. But as always, you need to remember to call the base class's implementation of those methods to make sure the rest of the system gets notified as expected.

GoChangedEventArgs is the class that represents an event for a document; it inherits from **System.EventArgs**. Besides remembering which document the event occurred for, it also remembers the kind of event, the new state value, and the old state value. The kind of event is described by the **GoChangedEventArgs.Hint** property, an integer. Some event hints, such as **GoDocument.ChangedPaperColor**, relate to the document itself. Other event hints apply to layers, such as **GoLayer.ChangedAllowDelete**, **GoLayer.InsertedObject**, and **GoLayer.RemovedObject**. Finally, other kinds of event hints pertain only to **GoObject**s, such as **GoLayer.ChangedObject**.

For some kinds of event hints, there is additional information that further describes the event. In particular, the **GoLayer.ChangedObject** event hint has an object specific sub-hint describing the exact kind of change and a previous value. For example, the **GoText** class has a **Bold** property. When the **Bold** property is changed, the setting method calls **GoDocument.RaiseChanged** with an event **Hint** of **GoLayer.ChangedObject** and a **GoChangedEventArgs.SubHint** of **GoText.ChangedBold**.

RaiseChanged is called just once for each separate change. Thus after generating a **Changed** event describing a **GoText** object whose **Bold** property changed, there is no need for another **Changed** event saying that the text object's document was changed also.

Each **GoChangedEventArgs** instance also holds any appropriate new and previous value information, so that the undo manager can record undo/redo information. This topic is covered in detail in Chapter 7.

The **GoDocument.IsModified** property is set to true by **GoDocument.OnChanged**. You will need to set this property to false whenever you store or reload your document.

Copying

You can add a copy of a collection of objects to a document by calling **GoDocument.CopyFromCollection**. This method makes copies of objects and maintains the relationships between them in the new copies. It also tries to preserve the layers of the original objects in the copies.

The way objects are copied is controlled by the **GoObject.CopyObject** methods of all the copied objects and by the **GoCopyDictionary**. A **GoCopyDictionary** is created (by **GoDocument.CreateCopyDictionary** if you don't create one and provide it) each time you want to copy one or more objects. It holds the results of the copying, mapping old objects to new objects. The copy dictionary, which is returned by **CopyFromCollection**, can be used afterwards to make changes to the copies or to select them.

The argument to **GoDocument.CopyFromCollection** is an **IGoCollection**. **GoDocument**, **GoLayer**, and **GoSelection** all implement **IGoCollection**, so it is easy to add a copy of all the objects in each of those kinds of collections into a document.

For most uses, the copy dictionary does not need to be initialized with any objects—the copy dictionary created by default is satisfactory. The copy dictionary is used to keep track of all copied objects, so that shared objects are not copied multiple times.

However, there are times when you don't want to create a new copy of an object because you want to use an already existing object in the destination document. Any references to the object in the source collection should be replaced by references to the existing destination object in the copied collection. You can achieve this effect by manually creating a **GoCopyDictionary** and initializing it so that the source object in question is mapped to the desired existing destination object. You then call **CopyFromCollection** passing in the initialized **GoCopyDictionary**. The copying process will notice that there is already a destination object in the copy dictionary, as if it had already been copied, and thus will not allocate a new object.

CopyFromCollection has some additional parameters that govern how it copies the objects in the argument collection. You can tell it to only copy objects whose **CanCopy** predicate is true. For objects that are not top-level, you can tell it to copy the object that would be dragged (presumably the parent node) rather than the individual child. And you can tell it to change the locations of the copied objects by a given offset.

If the original object belonged to a **GoLayer**, **CopyFromCollection** tries to add the copy in the destination document's layer that has the same identifier. If no such layer can be found, it is added to the **GoDocument.DefaultLayer**.

If you just want to add to a document a copy of a single object, you can use the **GoDocument.AddCopy** method, which just calls **CopyFromCollection**.

Persistence and Serialization

The built-in **GoDocument** and **GoObject** classes are **Serializable**. You should use serialization for short-term persistence and communication using the same version of the Go library. Northwoods does not recommend using serialization for long-term persistence to save diagrams that the user has edited. Besides the incompatibilities that arise when you change your application, serialized documents often contain much information that really should not be stored, because they describe the visual representation of the information rather than the abstract information that really matters.

For longer persistence, you will typically be loading from and storing into an existing database or file. Your code, which may include your **GoDocument** subclass and perhaps your **GoObject** subclasses, will then be responsible for transforming the real information into a network of **GoObject**s. Any user driven or programmatic changes to these objects must then be transformed back into the database's representation of the information. If the document permits any independent changes to the underlying database, you will need to be notified of those changes so that you can keep your document, and thus your views, up-to-date.

Some of the sample applications demonstrate how to read and write graphs to a simple XML file format.

Serialization, however, is used for cut/copy/paste operations to copy to the clipboard or to paste from the clipboard. It is also used for saving session state in GoDiagram Web. The data format is specified by **GoDocument.DataFormat**, and defaults to the full qualified name of the **GoDocument** type.

For copy and paste to work, you must make sure your **GoDocument** and **GoObject** derived classes have the **Serializable** attribute:

VB.NET: `<Serializable()> Public Class TestNode`

C#: `[Serializable] public class TestNode`

Furthermore you must make sure your fields are all **Serializable**. If they cannot be serialized, you can declare each field to be **NonSerialized**:

VB.NET: `<NonSerialized()> Private myPath As GraphicsPath = Nothing`

C#: `[NonSerialized] private GraphicsPath myPath = null;`

You will then need to make sure your code can handle a nothing/null value for this field—when the object is deserialized this field will get the default value for its type.

Sometimes you will have a copy and paste error because you forget to mark some classes or their fields with the appropriate attributes. One way to discover what you missed is create a document with instances of all kinds of objects in it, and

then to call the **GoDocument.TestSerialization()** method and note what exceptions occur. That method serializes and deserializes to an in-memory stream. You can do the same to and from a file:

C#:

```
Stream ofile = File.Open("test.graph", FileMode.Create);
IFormatter oformatter = new BinaryFormatter();
oformatter.Serialize(ofile, myView.Document);
ofile.Close();
Stream ifile = File.Open("test.graph", FileMode.Open);
IFormatter iformatter = new BinaryFormatter();
GoDocument doc = iformatter.Deserialize(ifile) as GoDocument;
ifile.Close();
goView1.Document = doc;
```

VB.NET:

```
Dim ofile As Stream = File.Open("test.graph", FileMode.Create)
Dim oformatter As IFormatter = New BinaryFormatter()
oformatter.Serialize(ofile, myView.Document)
ofile.Close()
Dim ifile As Stream = File.Open("test.graph", FileMode.Open)
Dim iformatter As IFormatter = New BinaryFormatter()
Dim doc As GoDocument = CType(iformatter.Deserialize(ifile), GoDocument)
ifile.Close()
goView1.Document = doc
```

Presumably you will be able to debug any exceptions that occur and figure out what source code changes are needed.

GoObject

GoObject is the superclass of all objects that can be contained in a **GoLayer**/**GoDocument** or a **GoView** and that can be displayed in a view.

GoObjects are very efficient in space and time compared with controls.

Bounding Rectangle and Location

Each **GoObject** has a size and a position, in document coordinates. There are many properties relating to the bounding rectangle. All ultimately depend on the **GoObject.Bounds** property. The properties are:

- **Bounds** – the bounding rectangle, a **RectangleF**

- **Position** – the top left corner of the bounds, a **PointF**

- **Size** – the dimensions of the bounds, a **SizeF**

- **Center** – the center point of the bounds, a **PointF**

- **Left** – the X coordinate of the left edge of the bounds

- **Top** – the Y coordinate of the top edge of the bounds

- **Width** – the horizontal distance between the left and right edges

- **Height** – the vertical distance between the top and bottom edges

- **Right** – the X coordinate of the right edge of the bounds, a **single** or **float** value that will be at least as large as the value of **Left**

- **Bottom** – the Y coordinate of the bottom edge of the bounds, a **single** or **float** value that will be at least as large as the value of **Top**

- **Location** – the customizable position of the object, a **PointF**

Override setting **Bounds** itself if you want to prevent certain bounds changes from happening at all.

If you want to constrain where an object can be moved, you may find it best to override **ComputeMove**, which is called by **DoMove**, which is called by **GoView.MoveSelection**.

If you want to constrain how an object is resized, you may find it best to override **ComputeResize**, which is called by **DoResize**, which is called by **GoToolResizing.DoResizing**.

Although normally one can think of the location of an object being the same as the **Position** value at the top left corner, that location might not be natural for some objects. Thus each object has its own notion of **Location**; by default this is the same as the **Position**. For example, **GoText** overrides **GoObject.Location** to use the text alignment in determining the natural position of the object, and a **GoBasicNode**'s location is the center of its **Shape** (typically an ellipse). Moving objects normally uses the **Location** rather than the **Position**.

There are a number of convenience methods for dealing with the standard nine spots of an object (corners, middles of sides, and center), and for repositioning two objects so that their particular user-specified spots coincide.

For example, the following code moves a label so that it is centered underneath an icon, touching it:

```
aLabel.SetSpotLocation(MiddleTop, anIcon, MiddleBottom)
```

The standard spots are:

- **Middle** – corresponding to the center of the bounding rectangle

- **TopLeft**

- **MiddleTop**

- **TopRight**

- **MiddleRight**

- **BottomRight**

- **MiddleBottom**

- **BottomLeft**

- **MiddleLeft**

The spot locations are also used to identify the standard resize handles. There are also **NoSpot** and **NoHandle** values for situations where there is no particular spot or handle.

When the **Location** is not the top-left corner of the bounding rectangle, changing either the **Size** or the **Position** will implicitly change the **Location**. To avoid having to make two changes when programmatically resizing an object, you can call the **GoObject.SetSizeKeepingLocation** method.

Other Properties

Each object has a number of boolean properties:

- **Visible --** can this object be seen in a view

- **Printable –** will this object be printed

- **Selectable --** can the user select this object, if also visible

- **Movable --** can the user move this object

- **Copyable –** can the user make a copy of this object

- **Resizable --** can the user change the size of this object

- **Reshapable –** can the user change the shape of this object

- **Deletable –** can the user remove this object from the document

- **Editable -**- can the user bring up a control to modify this object

- **AutoRescales –** if the size is changed, should it scale all its parts appropriately (for **GoText**, this means choosing a new **Font** size)

- **ResizesRealtime –** can the user see the object dynamically change size during resizing, instead of resizing a simple rectangle

- **Shadowed –** should this object be painted with a drop shadow

- **InvalidBounds –** should getting the **Bounds** call **ComputeBounds** in order to determine the correct bounds

- **SkipsUndoManager –**instructs the undo manager to stop recording information from events for this object

- **DragsNode –** whether this selected object, when moved, should move the parent node (or top-level object) instead

- **Initializing** – whether the object is in the process of being initialized, copied, or being undone or redone
- **BeingRemoved** – true during the process of removing an object from a layer or group

Not all kinds of objects support all of these properties, but almost all do.

Remember that properties such as **Selectable** and **Movable** just control the standard built-in behavior that Go views allow the user to do interactively using the mouse and/or key commands. You can always select or move objects programmatically, regardless of the property values, by explicitly calling methods such as **GoView.Selection.Add** and setting **GoObject.Position**.

Copying

If you want to add a copy of an object to a document, you can call **GoDocument.AddCopy**. If you want to make a copy of a single object without adding it to a document, you can call **GoObject.Copy().** This method just uses an instance of a standard **GoCopyDictionary**, in case there are references between child objects.

CopyObject is a method that is called by the copying process to provide a standardized way of transferring information to copies of objects. As you add fields to your subclasses, you will want to make sure the fields are copied appropriately when the object is copied. Your override of **CopyObject** should first call **base.CopyObject**; you can then modify the fields of the returned value.

The default implementation of **CopyObject** uses **MemberwiseClone** to make a copy of the object, which will automatically copy the values of all of the fields. It does not call a zero-argument constructor, which might not exist for the class. However, this can lead to unintended sharing of objects for fields that are references. It is important to make sure that you explicitly copy such field values in a way that is safe for your intended usage.

For example, the **GoStroke** class has a field that is an array of points. The **GoStroke.CopyObject** method explicitly makes a copy of that array so that modifying the points of the original stroke does not modify the copy, and vice-versa.

VB.NET:
```vbnet
Public Overrides Function CopyObject(ByVal env As GoCopyDictionary)
      As GoObject
  Dim newobj As GoStroke = CType(MyBase.CopyObject(env), GoStroke)
  If Not newobj Is Nothing Then
    newobj.myPoints = CType(myPoints.Clone(), PointF())
    ' . . .
  End If
  Return newobj
End Function
```

C#:
```csharp
  public override GoObject CopyObject(GoCopyDictionary env) {
    GoStroke newobj = (GoStroke)base.CopyObject(env);
    if (newobj != null) {
      newobj.myPoints = (PointF[])myPoints.Clone();
      . . .
    }
    return newobj;
  }
```

The copy dictionary argument to **CopyObject** is used to keep track of original objects and their copies.

CopyObject should return nothing/null when an object should not be copied. It should also return nothing/null if a copy is not necessary because the object is present in the copy dictionary. Only when a new copy has been allocated should **CopyObject** return a non-null value.

Another kind of copying occurs during serialization/deserialization going to and from the clipboard. Your **GoObject** classes must be **Serializable**. See the discussion about serialization and persistence of **GoDocument**s, earlier in this chapter.

Ownership

Most **GoObject**s should either belong directly to a **GoLayer** as a top-level document object or to a **GoGroup** that belongs to a layer/document. In either case the **Document** property value is this document and the **Layer** property value is the layer within the document. For child objects of groups, the **Parent** property value will be that **GoGroup** instead of null/nothing.

Occasionally some objects will properly belong to a **GoView** instead of to a **GoDocument**, because they really represent part of the "view" of the document and not of the document itself. Predefined cases include selection handles (**GoHandle**) and the in-place text editor. The size and position of view objects are in document coordinates. View objects have a **Layer** property value that is a layer in a view rather than a layer in a document.

Event Handling

When an object is changed, a **GoChangedEventArgs** with a **GoLayer.ChangedObject** hint is passed to all **GoDocument.Changed** event handlers. As you define subclasses with additional properties or other state, you will need to remember to raise this event. To do so it is easiest to call the **GoObject.Changed** method after the object's state changes, because it can take care of calling **GoDocument.RaiseChanged** for you. You should call **GoObject.Changed** only if the property value or object state really has changed.

A **GoChangedEventArgs** instance has a subhint value that is useful in identifying the kind of change that occurred for that subclass of **GoObject**. For example, a call to set the **GoObject.Visible** property will result in a call to

```
Changed(ChangedVisible, 0, old, NullRect, 0, value, NullRect)
```

This call passes along the subhint (**ChangedVisible**), and the old and new values. This additional information is important for optimizing update behavior and supporting undo and redo. Note that for efficiency (to avoid unnecessary heap allocation from boxing) the old and new values can be passed along as integer, **Object**, or **RectangleF** values.

The help file documentation for **GoObject.Changed** lists all of the predefined hint values for all of the Go object classes.

In addition to raising the **GoDocument.Changed** event, if you want to support undo and redo, you will need to make sure your **GoObject** subclass also handles new properties correctly in the **ChangeValue** method. For more about undo and redo, see Chapter 7.

Whenever any object is added or removed from a document or a view, it raises a **Changed** event with a **GoChangedEventArgs** with the appropriate **GoLayer.InsertedObject** or **GoLayer.RemovedObject** event hint for the corresponding **GoDocument** or **GoView**.

As you define your own subclasses, you can provide customized default behaviors for responding to various events. **GoObject** instances do not have their own individual events and event handlers because it is assumed that most of the objects of a certain class in a diagram want to behave the same way. This is unlike the situation where one expects to add controls to a form without subclassing and yet have radically different behaviors for each control. For controls, the overhead of having individual event handlers for each object is acceptable; it is not acceptable for **GoObject**s, where you may well have thousands of objects in a window at once.

The standard "event" handling methods are:

- **Paint --** render this object using a **Graphics**; if this method draws beyond the bounding rectangle, be sure to override **ExpandPaintBounds** correspondingly.

- **Pick** – is this object (or perhaps a selectable parent) under a given point

- **OnLayerChanged --** the object has just been added to or is about to be removed from a layer in a document or view

- **OnParentChanged** – the object has just been added to or is about to be removed from a group

- **OnBoundsChanged --** the object has changed size and/or position

- **OnSingleClick --** the user just clicked on this object

- **OnDoubleClick --** the user just double-clicked on this object

- **OnContextClick --** the user just right-mouse clicked on this object

- **OnHover** [GoDiagram Win only] – the mouse has been resting at the same point over this object for a length of time determined by the **GoView**

- **OnMouseOver** [GoDiagram Win only] **--** the user just moved the mouse over this object without holding a mouse button down

- **OnEnterLeave** – the mouse has either just entered or just left this object, either during a mouse-over or while dragging the selection

- **OnSelectionDropReject** – the user is dragging the view's selection on this object—return true to disallow the drop

- **OnSelectionDropped** – the user has just dropped the view's selection onto this object

- **GetToolTip --** return a string to display in a tool tip (defaults to nothing/null)

- **GetCursorName** – return the name of a cursor to be shown for the mouse pointer

- **OnGotSelection --** this object just got added to some view's selection; typically this will call **AddSelectionHandles** on this object's **SelectionObject**, which in turn will normally call **CreateBoundingHandle** or **CreateResizeHandle**.

- **OnLostSelection --** this object just got removed from some view's selection; typically this will call **RemoveSelectionHandles** on this object's **SelectionObject**.

- **DoMove** – the user is moving this object interactively; normally this will call **ComputeMove**.

- **DoResize --** the user is resizing this object interactively; normally this will call **ComputeResize**.

- **DoBeginEdit** [Windows Forms only] – start the user editing this object interactively; typically this will call **CreateEditor**.

- **DoEndEdit** [Windows Forms only]– stop editing, if any object editor is in progress

Many of these methods are called in response to user interactions with the **GoView**, and are related to events generated by the view. For more information, see Chapter 5.

Other Notifications of Object Changes

When you want to do something when certain changes happen to objects, you can override the **GoObject.Changed** method to notice everything, or you can override the setting of certain properties. If you don't want to override a method or property, or if you can't because you cannot define a subclass of an object, you can either add a **GoDocument**.**Changed** event handler (as discussed earlier), or you can add an observer to a particular object.

A **GoObject** can easily notice changes to another object by declaring itself to be an observer of the other object. You can do this by using the **AddObserver** and **RemoveObserver** methods. Note that an observer and the observed object must both be **GoObject**s—this lets them take part in copying and persistence very naturally.

When an observed object's **GoObject.Changed** method is called it first calls its **Document**'s (or **View**'s) **RaiseChanged** method for performing the standard updating. Then it calls the **GoObject.OnObservedChanged** method of each observer, passing it the changed object and all the same argument values that the **Changed** method got.

This mechanism is sometimes used to keep track of a particular property of a particular child object of a **GoGroup** by adding the parent to the child's list of observers and by overriding the group's **OnObservedChanged** method to look for the desired change subhint corresponding to that property and for that particular child object. You should try to avoid using the observer mechanism to keep track of unrelated objects, or when the relationship between the objects might change (such as when an object might be removed from the group).

GoObject-inheriting Classes

Here is a class hierarchy diagram for the main **GoObject**-inheriting classes:
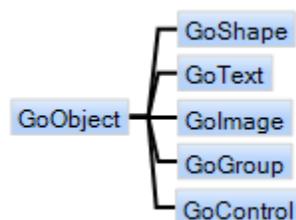
The principal subclasses of **GoObject** include **GoShape**, **GoText**, **GoImage**, and **GoGroup**. These are discussed in the following sections.

GoShape

Shapes include both closed and filled two-dimensional objects and open/unfilled (linear) objects such as **GoStroke**s. Strokes are multi-segmented straight or curved lines. Strokes can also have arrowheads.

Most shapes, though, are things like diamonds, ellipses, polygons, rectangles, rounded rectangles, triangles, and pie slices. Some classes, such as **GoCube** and **GoCylinder**, provide a simulated view of a 3D shape.
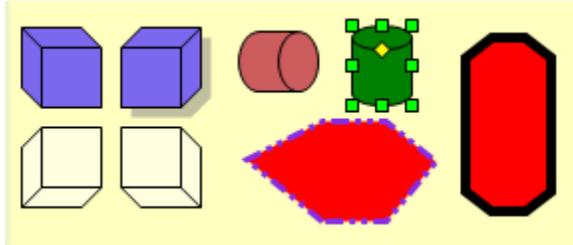
Here's the class hierarchy starting with **GoShape**:

Each **GoShape** has a **Brush** and a **Pen** to specify how to fill the inside of the shape and how to draw the outline of the shape. Because **GoStroke**s are "open" shapes, the **Brush** specifies whether and how to fill in an arrowhead(s).

You will typically set the **GoShape.BrushColor**, **BrushStyle**, and **BrushForeColor** properties to fill the shape with the brush that you want. Similarly, you can set the **GoShape.PenColor** and **PenWidth** properties to control the most commonly set **Pen** properties.

But you can also construct your own **Pen** and **Brush** values. This is useful when you want a dotted pen, or a texture or gradient brush. Be sure to finish setting them up the way you want before you set the **GoShape.Pen** or **GoShape.Brush** properties, because you may not change a **Pen** or **Brush** after you have assigned it as a property value.

```
Pen p = new Pen(Color.DarkTurquoise, 5);
p.DashStyle = DashStyle.Dash;
s.Pen = p;
```

Note: fancy pens and hatch or gradient brushes are not supported by the .NET Compact Framework (GoDiagram Pocket).

Constructing gradient brushes can be somewhat more complicated. For convenience **GoShape** defines several **Fill…** methods that provide common effects. For example:

```
roundrect.FillSimpleGradient(Color.Blue)
```



```
roundrect.FillMiddleGradient(Color.Blue)
```



```
roundrect.FillHalfGradient(Color.Blue)
```



```
roundrect.FillShadedGradient(Color.Blue)
```



```
roundrect.FillSingleEdge(Color.Blue)
```

```
roundrect.FillDoubleEdge(Color.Blue)
```

```
roundrect.FillShapeFringe(Color.Blue)
```

```
roundrect.FillShapeGradient(Color.Blue)
```

```
roundrect.FillShapeHighlight(Color.Blue)
```

These methods also have overloads that take the "other" color and, for the linear gradients, the direction of the gradient.

```
roundrect.FillSimpleGradient(Color.Red, GoObject.MiddleLeft)
```

```
roundrect.FillSimpleGradient(Color.White, Color.LightBlue,
GoObject.TopLeft);
```

```
roundrect.PenColor = Color.LightBlue;
```

```
roundrect.FillSingleEdge(Color.Red, Color.Orange,
GoObject.MiddleTop)
```

```
roundrect.BrushMidFraction = 0.4f
```

```
ellipse.FillShapeGradient(Color.DarkKhaki, Color.Khaki)
```
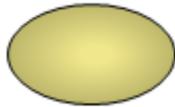


```
triangle.FillShapeGradient(Color.Navy, Color.SkyBlue)
```
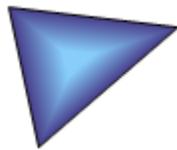

A **GoTextNode** using a **GoRoundedRectangle** as its **Background** shape, could appear with a linear gradient as follows:

```
textnode.Label.TextColor = Color.White;
```

```
textnode.Shape.FillHalfGradient(Color.Black);
```
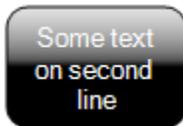

Here's an example showing a path gradient:

```
textnode.Shape.FillShapeHighlight(Color.Blue, Color.White)
```

```
textnode.Shape.BrushPoint = new PointF(0.2f, 0.2f)
```

```
textnode.Shape.BrushFocusScales = new SizeF(0.9f, 0.85f);
```
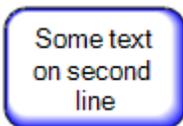

Please note that **PathGradientBrush**es only work well with simple convex shapes.

Once you have created a gradient brush and assigned it to the **GoShape.Brush** property by calling one of the **Fill…** methods, you can get or set the colors that the brush uses, without having to reconstruct the brush explicitly.

Brush and Pen Properties

Setting the **GoShape.BrushStyle** property causes a new brush to be created and assigned as the value of **GoShape.Brush**. The type of the brush, and some of it basic characteristics, are determined by the particular **GoBrushStyle** enum value.

There are three color properties relevant to brushes on **GoShape: BrushColor**, **BrushForeColor**, and **BrushMidColor**. Please note that setting these properties, particularly **BrushForeColor** and **BrushMidColor**, may have no effect until the shape has the appropriate brush style. You can also set the **BrushMidFraction** property to control the fractional distance at which the middle gradient color is drawn, for those **GoBrushStyle**s that display three colors.

For linear gradients, the **BrushPoint** property lets you set the end point for the gradient; the **BrushStartPoint** property specifies the starting point. These **PointF** values take fractional single floating point numbers, typically between 0 and 1. These fractions are scaled up by the width and by the height of the shape to determine the actual point within the shape. You can also use values a little bit less than zero or a little bit larger than 1 to specify points just outside of the shape's bounding rectangle.

For path gradients, the **BrushPoint** property specifies the center point of the focus area. It too uses normalized fractional values based on the size of the shape. But path gradients also allow you to control the size of the focus area that is displaying the **BrushColor** -- use the **BrushFocusScales** property. It is of type **SizeF**, and its values are also fractional values of the width and height of the shape.

You can also set or get the **GoShape.PenColor** and **PenWidth** properties, as alternate and more convenient ways of customizing the **Pen**. Setting the **PenColor** to **Color.Empty** will just set the **Pen** to **null** – no shape outline is drawn. Note also that a pen width of zero has the convention in GDI+ of drawing as a single-pixel-wide pen, regardless of the **GoView.DocScale**.

Since many of the node classes offer a **Shape** property for accessing its shape object as a **GoShape**, you can use this property to easily customize the appearance of nodes. For example:

```
GoTextNode n = new GoTextNode();
n.Text = "a GoTextNode";
n.Label.Bold = true;
n.Label.FontSize = 14;
n.Shape.BrushColor = Color.Tomato;
n.Shape.BrushForeColor = Color.Bisque;
n.Shape.BrushStyle = GoBrushStyle.HatchHorizontalBrick;
n.TopLeftMargin = new SizeF(15, 10);
n.BottomRightMargin = new SizeF(15, 10);
doc.Add(n);
```

```
GoBasicNode n = new GoBasicNode();
n.Text = "a GoBasicNode";
n.LabelSpot = GoObject.Middle;
n.MiddleLabelMargin = new SizeF(20, 30);
n.Shape.FillShapeFringe(Color.Violet);
doc.Add(n);
```


Dynamic Brushes

GoShape will automatically rescale its gradient brush to fit the size of the shape. However, there can be situations where you really need to generate a brush dynamically. For example, when you want a fringe of a constant width regardless of the size or aspect ratio of the shape, you will need to produce a new brush each time the shape changes size.

The **PathGradientRoundedRectangle** example class in the Demo1 sample demonstrates this technique. It overrides the **GoShape.Brush** property getter to return a newly created **PathGradientBrush** each time. To optimize the work, the resulting brush is cached in a field which is cleared whenever the **Bounds** property is changed, including during an undo/redo operation in **ChangeValue**.

A **PathGradientRoundedRectangle**, with a White **BrushColor** and a Gray **BrushForeColor**:

Although most shapes will be instances of **GoRectangle** or **GoEllipse**, you may find it convenient to use the **GoDrawing** class to get some of the common shapes without painstakingly initializing a **GoPolygon**.

GoDrawing and Predefined Figures

The most general **GoShape** class is **GoDrawing.** This shape class is like **GoPolygon** in supporting an arbitrary number of segments, but allows one to mix straight and Bezier curve segments, and can have any number of separate open or closed figures.

For example, to create a "Rounded I-Beam" shape that looks somewhat like a capital "I" with concave curves:

```
GoDrawing s = new GoDrawing();
s.StartAt(0, 0);
s.LineTo(100, 0);
s.CurveTo(50, 25, 50, 75, 100, 100);
s.LineTo(0, 100);
s.CurveTo(50, 75, 50, 25, 0, 0);
```

As another example, to create a heart shape:

```
GoDrawing s = new GoDrawing();
s.StartAt(50, 25);
s.CurveTo(50, 0, 100, 0, 100, 30);     // Top right
s.CurveTo(100, 50, 50, 90, 50, 100);   // Bottom right
s.CurveTo(50, 90, 0, 50, 0, 30);       // Bottom left
s.CurveTo(0, 0, 50, 0, 50, 25);        // Top left
```

GoDrawing also supports rotation. You can either set the **Angle** property, or you can call **Rotate** to incrementally change the angle about an arbitrary point.

You can also flip the drawing about either the vertical or the horizontal axis.

Predefined **GoDrawing** shapes are defined by the **GoFigure** enumeration. For example:

```
GoDrawing s = new GoDrawing(GoFigure.Cloud);
s.Bounds = new RectangleF(10, 10, 120, 80);
s.BrushColor = Color.WhiteSmoke;
```

produces something that might look like:

Caution: the exact appearance of these predefined drawings may change in future versions.

A number of the node classes also define constructors that take a **GoFigure** parameter and offer a **Figure** property for convenience in setting the **GoDrawing.Figure** if the shape is an instance of **GoDrawing.**

Caution: setting the **Figure** property of a node class if the node's shape is not an instance of **GoDrawing** will have no effect. Since the default kind of object for most of the node classes is *not* a **GoDrawing**, due to efficiency considerations, you have to make sure the shape is an instance of **GoDrawing** before setting the **Figure** property.

Here's a listing of all of the **GoFigure**s that are currently defined. Again, this list may change in the future, as may the appearance of the individual figures.

None	Line1	Line2	Curve1	Curve2	Curve3	Curve4	Triangle

FivePointedStar	SixPointedStar	SevenPointedStar	EightPointedStar	NinePointedStar	TenPointedStar	FivePointedBurst	SixPointedBurst

FramedRectangle	HalfEllipse	Heart	Spade	Club	HourGlass	Lightning	Parallelogram1

ThickX	ThinCross	ThinX	Trapezoid	YinYang	Peace	NotAllowed	Fragile

StopSign	LogicImplies	LogicIff	LogicNot	LogicAnd	LogicOr	LogicXor	LogicTruth

DoubleArrow	DoubleEndArrow	IBeamArrow	Pointer	RoundedPointer	SplitEndArrow	SquareArrow	Cone1

Prism2	Pyramid1	Pyramid2	Actor	Alternative	Card	Collate	Connector

Display	DividedEvent	DividedProcess	Document	ExternalOrganization	ExternalProcess	File	Gate

MagneticTape	ManualInput	ManualLoop	ManualOperation	Merge	MessageFromUser	MessageToUser	MicroformProcess

PrimitiveFromCall	PrimitiveToCall	Procedure	Process	SequentialData	Sort	Start	StoredData

NandGate	NorGate	OrGate	XnorGate	XorGate	Capacitor	Resistor	Inductor

IrritationHazard	ElectricalHazard	FireHazard

Diamond	Pentagon	Hexagon	Heptagon	Octagon	Nonagon	Decagon	Dodecagon
SevenPointedBurst	EightPointedBurst	NinePointedBurst	TenPointedBurst	Circle	Cloud	Crescent	Ellipse
Parallelogram2	Rectangle	RightTriangle	RoundedIBeam	RoundedRectangle	Square	SquareIBeam	ThickCross
GenderMale	GenderFemale	PlusLine	XLine	AsteriskLine	CircleLine	Pie	PiePiece
LogicFalsity	LogicThereExists	LogicForAll	LogicIsDefinedAs	LogicIntersect	LogicUnion	Arrow	Chevron
Cone2	Cube1	Cube2	Cylinder1	Cylinder2	Cylinder3	Cylinder4	Prism1
CreateRequest	Database	DataStorage	DataTransmission	Decision	Delay	DirectData	DiskStorage
Input	Interupt	InternalStorage	ISOProcess	Junction	LinedDocument	LoopLimit	MagneticData
MicroformRecording	MultiDocument	MultiProcess	OfflineStorage	OffPageConnector	Or	Output	PaperTape
Subroutine	Terminator	TransmittalTape	AndGate	Buffer	Clock	Ground	Inverter
ACvoltageSource	DCvoltageSource	Diode	Wifi	Email	Ethernet	Power	Fallout

GoText

Text strings are displayed by the **GoText** class. There are many properties that help determine the appearance and behavior of a **GoText** object:

- **Text** – the string to be displayed

- **FamilyName** – the string name of the font family to be used, such as `"Microsoft Sans Serif"`

- **FontSize** – the point size specifying the height and width of the characters, such as 10

- **Alignment** – how each line of text is aligned within the whole text object, such as **GoObject.Middle** for centered text; this also determines the **Location** for the object

- **TextColor** – the color for the characters, such as **Color.Black**

- **BackgroundColor** – the color for the background behind the text, such as **Color.White**

- **TransparentBackground** – if true, the background color is not painted; otherwise the whole text object is filled with the background color

- **Bold** – whether the text is in a bold style

- **Italic** – whether the text is in an italicized style

- **Underline** – whether the text is underlined

- **StrikeThrough** – whether the text appears "crossed out"

- **Bordered** – whether the text has a rectangle drawn around it, in the **TextColor**

- **Multiline** – whether embedded carriage-return/newline character sequences force a line break in the display of the text string

- **AutoResizes** – whether the size of the text object is automatically adjusted as the text string is changed

- **StringTrimming** – how the text is abbreviated when **AutoResizes** is false

- **Clipping** – whether the text drawing is clipped to the bounds of the text object

- **BackgroundOpaqueWhenSelected** – whether selecting a text object causes the background to be displayed (**TransparentBackground** set to false) instead of getting selection handle(s) as most objects normally do.

- **Wrapping** – whether to automatically insert line breaks even when there is no newline character embedded in the string

- **WrappingWidth** – when **Wrapping** is true, specifies the width at which text will be wrapped to the next line, in document coordinates

- **EditableWhenSelected** – when true, permits editing of the text only when it is part of an object that was selected before it was clicked

- **EditorStyle** [Windows Forms only] – this controls the kind of **Control** used to implement the in-place text editor

When a **GoText** object is constructed, the **FamilyName** and **FontSize** properties default to the values of the shared/static variables **GoText.DefaultFontFamilyName** and **GoText.DefaultFontSize**. By default, text objects are not **Resizable** and have a **TransparentBackground**. They support only single lines of text and do not wrap or clip.

The **AutoResizes** property, which defaults to true, causes the text string to be remeasured each time the string value is changed and the **GoText**'s **Bounds** property to be updated accordingly. The **Location** (as determined by the **Alignment**) will stay the same, but the width and height will match the dimensions of that text string, in the given font and style. If you set **AutoResizes** to false or if you explicitly change the **Size** of the text object, you run the risk of painting beyond the bounds of the text object, which will result in improper updates of the view. In this case it is wise to set the **Clipping** property to be true, to make sure that the text is not drawn beyond the bounds of the object. The **Clipping** property defaults to false for performance reasons.

The **GoText.BackgroundOpaqueWhenSelected** property determines how a selected text object appears by controlling the transparency of the text's background instead of adding selection handles.
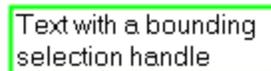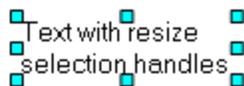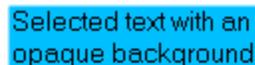
A **GoText** object whose **Shadowed** property is true will produce a rectangular shadow if **TransparentBackground** is false and will produce an exact shadow of the text characters if **TransparentBackground** is true.

For improved performance the **Paint** method calls the **PaintGreek** method to allow it to decide on simpler renditions of the text at small scales. The standard implementation uses the **GoView.PaintNothingScale** and **GoView.PaintGreekScale** to decide if the text should be painted at all or if it should just be drawn as a single line.

With Windows Forms users can edit text in-place. If the **Editable** property is true, then a single click on the text object will invoke **DoBeginEdit** to create and display a **TextBox** control. The **Multiline** property determines the behavior of the Enter key. When **Multiline** is true, the **TextBox** accepts the Enter key as inserting a carriage-return/newline; when false, the Enter key calls **DoEndEdit** to finish editing, resulting in a modified **GoText.Text** string value. In either case the Escape key calls **DoEndEdit** without changing the string value.

For text objects that represent integers, you can set the **EditorStyle** property to **GoTextEditorStyle.NumericUpDown**. In this case **GoText.CreateEditor** will bring up a **NumericUpDown** control, limited to a range of integers specified by **GoText.Minimum** and **GoText.Maximum**. If you set the **EditorStyle** property to **GoTextEditorStyle.ComboBox** and you can set the **GoText.Choices** property to a list of items that will be presented in the drop-down list of a **ComboBox** control. The **GoText.DropDownList** property specifies whether the user is allowed to type arbitrary text in the **ComboBox**.
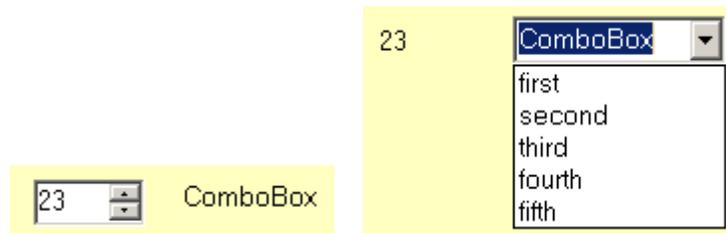
GoImage

Various kinds of images, such as bitmaps, WMF files, GIF files, JPEG files, and icons are displayed using the **GoImage** class. The images can be kept as files or can be stored in resources, either separately or as part of **ImageList**s. Note that **ImageList** is only available when using Windows Forms.

Properties:

- **Image** – the underlying **Image** object

- **ResourceManager** – the **ResourceManager** in which to look up **Image** values by name

- **Name** – either the name of the image resource in the **ResourceManager** or the filename on disk

- **NameIsUri** – the name is not a pathname for a disk file, but is a URI that a **WebClient** can use to find an image

- **ImageList** – the **ImageList** containing **Image**s indexed by integer [Windows Forms only]

- **Index** – if non-negative, the integer index of the desired **Image** in the **ImageList** [Windows Forms only]

- **Alignment** – where the actual image is drawn within the whole **GoImage** object; this also determines the **Location** for the object

- **AutoResizes** – whether the size of the **GoImage** object is automatically adjusted as the **Image** is changed

The **GoImage** constructor creates an image object that is not **Reshapable** by default, thereby maintaining its aspect ratio when resized by the user

The initial value of the **ResourceManager** property is the value of **GoImage.DefaultResourceManager**, which itself is initially nothing/null.

The shadow of a **GoImage** is drawn in the same shape as the non-transparent parts of the **Image**.

GoImage keeps a static/shared hashtable of cached images. This helps reduce memory consumption, for example when creating multiple nodes that all display the same image. You can clear this cache by calling **GoImage.ClearCachedImages**. However, no existing **GoImage**s will change appearance until you call **GoImage.UnloadImage**, which will cause **LoadImage** to reload a new image from a **ResourceManager**/**ImageList**/disk file when the **GoImage** is painted in a **GoView**.

You should override **GoImage.LoadImage** if you have alternate means of getting an **Image** in memory and you depend on serialization. Setting the **Image** property works, but the **Image** is not serialized. When a **GoImage** is serialized and deserialized, it depends on the **LoadImage** method to reproduce the **Image**. If **LoadImage** fails, no image will show in the view. You can interpret the **Name** and **Index** properties however you wish, and of course you can add whatever serialized fields you need to ensure your override of **LoadImage** works.

GoGroup

GoGroup implements the concept of a "group" of objects that can be manipulated together. These objects will **not** also be contained directly by any layer or by other groups; **GoGroup** and **GoLayer** will enforce this policy.

GoGroup is a subclass of **GoObject**, which means that groups can contain other groups. This is the Composite pattern. Using this mechanism, an object hierarchy can be created.

GoGroup also implements the **IGoCollection** and **IList** interfaces using an **ArrayList**. Unlike **GoLayer** and **GoDocument**, the objects in a group maintain a particular order. Use the **InsertAfter** and **InsertBefore** methods to add an object into a group at a particular position relative to other children in the group. **Add** always inserts the object at the end of the list, so that it always appears in front of other children.

You can use the **GoGroup.AddCollection** method for adding a collection of objects to be immediate children of a group. This method can even move objects from within other **GoGroup**s or top-level objects, without disconnecting any links as would normally happen if objects are first **Remove**d and then **Add**ed.

The coordinates for objects within a group are kept in document coordinates; they are not relative to the position of the group.

A group does not really have its own independent bounding rectangle. Instead the bounding rectangle is really the bounding rectangle for all of the children. In fact the **Bounds** property is not meaningful when there are no objects in a group.

Most of the **GoGroup** methods just iterate over the child objects, performing the appropriate operation. **Paint**, for example, just calls **Paint** on each visible child.

When you add an object to a group, you will normally make that child object not **Selectable**.

When a child object is not **Selectable**, the selection mechanism will handle a user mouse click on the child object by trying to select its parent group. If that group is **Selectable**, it is selected; otherwise the selection mechanism continues trying up the chain of parent groups.

When a child object is **Selectable**, it can be selected as if it were an independent object. Both it and its parent group and any sibling objects can belong to the selection simultaneously. When the user then drags such selected children, the behavior depends on the object's **DragsNode** property. If true, dragging the child will drag the parent **IGoNode** instead (not just the parent **GoGroup**, in case the groups are deeply nested). If **DragsNode** is false, the user can drag the object around and any effects on the parent group are determined by how that group's **GoGroup.LayoutChildren** method behaves.

If a **GoGroup** object is removed from a layer, all of its children are also removed. However, setting one of the properties such as **Visible** or **Deletable** does not cause the same properties to be set on any of the children. Nevertheless a child object whose **Visible** property is true will not be viewable by the user if its parent group's **Visible** property is false.

It is fairly common to want to refer to a particular child object for various reasons, such as wanting to change its appearance or when laying out the position of certain objects relative to each other. The most efficient way to get and retain such references is to define a subclass and add a field that refers to the child object. Nearly all of the predefined node classes and most of the example classes do this.

However, you might not want to bother defining a subclass of a group or node, particularly when there is no method that you need to override. Another way of keeping track of particular child objects is to associate a name with them, by calling the **GoGroup.AddChildName** method. You can recover the child reference by calling **FindChild** or (in C#) using a String indexer. Many of the children of the predefined node classes already have such names—the names are the same as the names of the properties that return the child.

Basically just after you construct, initialize, and add a new child object to a group, you can call **AddChildName** so that **FindChild** will return that child.

Bounds Management

Setting the **Bounds** property changes any object's position and size. Such a change will also invoke the **OnBoundsChanged** method and all document **Changed** event handlers with a **GoChangedEventArgs** holding a **ChangedBounds** subhint. Remember that these methods get called *after* the bounding rectangle has been changed.

The default behavior implemented by the **OnBoundsChanged** override for **GoGroup** calls **RescaleChildren** to move all the children and resize them by the same horizontal and vertical scales that the whole group is being resized. It then calls **LayoutChildren**, which by default does nothing, since it has no object specific knowledge about how to reposition the children in the desired manner.

For groups that include text strings the built-in resize may not appropriate, since the user probably does not want to change the size of the text. In such cases it is better to either turn off the **AutoRescales** property on the (text) object or to manage the layout of the group's children explicitly. It is fairly common to set **AutoRescales** to false for certain group children, especially text. But it is also common to override **LayoutChildren** in order to re-position and perhaps re-size the group's children to maintain a certain appearance.

When neither the width nor the height of the whole group has been changed, it is convenient to use the **MoveChildren** method for moving all of the group's children. In fact, **GoGroup.OnBoundsChanged** only calls **MoveChildren** when the new group size is the same as the original size.

When a group's child is changed by setting its **Bounds** property, the parent group is notified by a call to **OnChildBoundsChanged**. This allows the group the opportunity to adjust its notion of its position and size, and to re-layout the children if desired. By default **GoGroup.OnChildBoundsChanged** just calls

LayoutChildren. Your individual group classes may wish to adjust the size and/or position of some of the other group children. But remember that the change was instigated by a change to a child, and not to the group as a whole. Be careful to avoid infinite adjustment loops or differing behaviors depending on the order of changes.

The argument to **LayoutChildren** will indicate which child, if any, had changed bounds; the argument will be nothing/null when called due to the whole group's bounds having changed.

You need to consider whether users trying to move or copy a child object should instead move or copy the parent. Because most children are not **Selectable** this is not an issue. But if they are selectable, your override of **GoGroup.LayoutChildren** will automatically reposition each of the children in the right place when the group is resized, which will keep the child in place! If you want to allow children to be selected and able to be moved on their own, you should make sure that the **LayoutChildren** method does not control their positioning.

On the other hand, if you want the children of a group to be individually selectable but you do not want the user to move them independently, you should set the **GoObject.DragsNode** property to true for each of these children. This will let a user's drag of a selected child drag the whole group.

If the object's shape isn't like the bounding rectangle, you may need to override **ContainsPoint** to improve picking, and override **GetNearestIntersectionPoint** to improve calculating link points for ports.

Kinds of Groups

Go provides many different kinds of predefined groups. Most are nodes, because they have ports and can be linked together -- see Chapter 6. Here's the class hierarchy starting with **GoGroup**:
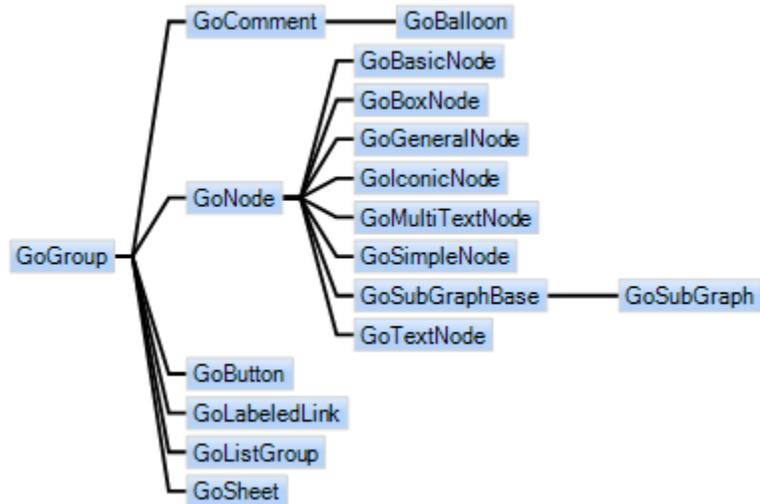
GoPort

GoPort acts as a connection point for **GoLink** objects. Each port has a collection of **GoLink**s that are attached to the port.

As with any class implementing **IGoPort**, each **GoPort** has two properties, an integer (**UserFlags**) and an object (**UserObject**), for your use. These properties can sometimes be handy to associate your application data with a port without having to define a new class inheriting from **GoPort**.

Appearance

By default a **GoPort** appears as an ellipse, but it can use any **GoObject** to control its appearance. **GoPortStyle** enumerates the predefined styles:

- **None** – draw nothing, but allow participation in linking

- **Object** – another object (a "Port Object") provides the representation using the port's bounds

- **Ellipse** – draw an ellipse (or circle)

- **Triangle** – draw a triangle "pointing" according to the value of the port's **ToSpot** property, as if the link were an arrow coming into the port

- **Rectangle** – draw a rectangle (or square)

- **Diamond** – draw a four-sided polygon with the vertices at the midpoints of the bounding rectangle's edges

- **Plus** – draw a "+"

- **Times** – draw an "x"

- **PlusTimes** – draw both a "+" and an "x" at the same spot

GoPort is a subclass of **GoShape**, so you can easily control the appearance of the non-**None**, non-**Object** ports by setting the **Pen…** and/or **Brush…** properties.

The following image shows two nodes, each with two ports. One port is diamond shaped, with a cornflower-blue brush. Another port is triangular, with no brush. Finally, two ports are ellipses, with a light green brush and no pen.
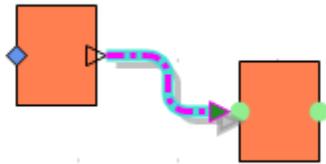
Ports can also share many Port Objects. Your application can, for example, pre-allocate several different **GoImage** instances corresponding to the kinds of states you want to display to the user. As each port changes state, you just need to set the **PortObject** property with the appropriate image. Because potentially many ports will share these Port Objects, they must not be part of any document or group or view. Before each Port Object is painted, its bounding rectangle will be set to the bounding rectangle of the port.

Linking Ports

For your application, some ports may be valid sources for links, some may be valid destinations, and some may be both or neither. It may be that some particular pairs of ports cannot have a valid new link between them. For example, you may want to avoid having two different links connecting the same two ports, or you may want to limit the number of links on a port to a certain number. The principal method that is called is **GoPort.IsValidLink**. It is responsible for deciding if it is OK for a user to draw a new link or reconnect an existing link to go between two particular ports.

The linking tool, **GoToolLinking**, uses the **CanLinkFrom**, **CanLinkTo** and **IsValidLink** methods to allow the particular port classes the ability to control whether the user can draw a link starting at a given port and ending at one.

GoPort also provides several properties that affect the behavior of those predicates:

- **IsValidFrom**
- **IsValidTo**
- **IsValidDuplicateLinks**
- **IsValidSelfNode**

- **IsValidSingleLink**

You can set the **IsValidFrom** and/or **IsValidTo** properties to false to cause the **CanLinkFrom** and **CanLinkTo** methods to return false. Other settable **GoPort** properties include **IsValidSelfNode** and **IsValidDuplicateLinks**, both used by **IsValidLink** to determine link validity. Normally a link is not allowed from a port to a port in the same node. Only when **IsValidSelfNode** is true for both ports may **IsValidLink** return true. Similarly, when a link already exists, a second link is not allowed from the same **FromPort** to the same **ToPort**. Only when **IsValidDuplicateLinks** is true for both ports may **IsValidLink** return true. Finally **IsValidSingleLink** permits the user to connect at most one link to a port.

GoPort.IsValidLink also looks at the port's **GoDocument.ValidCycle** property to decide if it needs to see if a cycle might result from connecting the proposed two ports.

You can also override the **CanLinkFrom** and/or **CanLinkTo** methods, as with the **LimitedPort** example port class. The following code imposes an optional maximum number of links for a port, based on a **MaxLinks** property that specifies a limit.

VB.NET:
```
Public Overrides Function CanLinkFrom() As Boolean
  Return MyBase.CanLinkFrom() AndAlso Me.LinksCount < Me.MaxLinks
End Function
Public Overrides Function CanLinkTo() As Boolean
  Return MyBase.CanLinkTo() AndAlso Me.LinksCount < Me.MaxLinks
End Function
```

C#:
```
public override bool CanLinkFrom() {
  return base.CanLinkFrom() &&
         this.LinksCount < this.MaxLinks;
}
public override bool CanLinkTo() {
  return base.CanLinkTo() &&
         this.LinksCount < this.MaxLinks;
}
```

Because ports have a size, the exact point at which a link should terminate may want to depend on the dimensions of the port. Furthermore it is common for there to be different points depending on whether the link is coming in or going out of the port or where the port is located relative to the rest of the node. This notion is supported by the **FromSpot** and **ToSpot** properties, which remember the object spots that links connected to this port should end at. The **GetLinkPoint** method is responsible for calculating this point; the default behavior depends on the **FromSpot** and **ToSpot** values.

Override the **GetLinkPoint** method to produce more sophisticated link appearances. Usually if the link direction for the port is on one side, the link point will be on the same side to avoid overlapping the link with the visual appearance of the port. Note that the link point need not be in the bounding rectangle of the port, although if it is too far away it might be confusing or disconcerting for the user.

If you expect the link point to vary dynamically, you may wish to specify **NoSpot** as the value for one or both of the **FromSpot** and **ToSpot** properties. In this case the **GetLinkPointFromPoint** method is called. By default this calls **GetNearestIntersectionPoint**. The argument specifies approximately where the link is coming from or going to. As a further convenience, **GetNearestIntersectionPoint**, when the port style is not **Style.Object**, uses the edge point of the Port Object that intersects the straight line from a point in the link's stroke to the center of the port. For example, **BasicNode** sets its port's **PortObject** to be its ellipse, which has the effect of ending links not at the port but at the outer edge of the ellipse.

Links that are connected to a port may be constrained to come into the port or come out of the port from certain directions. **GetLinkDir** is responsible for determining the direction. The standard directions correspond to the spot locations. If the spot is **Middle** or **NoSpot** you may want to override this method to return the desired direction.

Navigating Links

Each port has a collection of links that are attached to the port. The links do not belong to the port–normally the links are top-level objects in a document. From a port you can iterate over all the links to get to all the ports connected by those links. For example, here is the code in the Family Tree example where the document is positioning all the "children" **PersonNode**s for a particular mother/father pair. All of the children are linked to the mother/father marriage at a "marriage port", here held in a variable named mp.

VB.NET:
```
' now look at each child
Dim childrect As RectangleF = mp.Bounds
Dim childlink As IGoLink
For Each childlink In mp.Links
  Dim childp As IGoPort = childlink.GetOtherPort(mp)
  Dim childnode As PersonNode
  childnode = CType(childp.GoObject.Parent, PersonNode)
  LayoutTree(childnode, childrect)
Next
```

C#:
```
// now look at each child
foreach (IGoLink childlink in mp.Links) {
```

```
    IGoPort childp = childlink.GetOtherPort(mp);
    PersonNode childnode = (PersonNode)childp.GoObject.Parent;
    LayoutTree(childnode, ref childrect);
}
```

This code iterates over the links at the `mp` port. It gets the port at the other end of the link. Then it gets the **PersonNode** for that other port by getting the port's parent group and assuming it is of the correct class. Finally it calls a method with that node representing the child.

If you only wish to look at links on a port going in a single direction, **GoPort.SourceLinks** returns an enumerable for iterating over only links coming in to the port. **GoPort.DestinationLinks** returns a similar enumerable for iterating over links leaving the port.

GoLink

GoLink is a **GoStroke** that connects two different **GoPort**s. Normally you create a link by allocating a new **GoLink,** setting both the "from" and "to" ports, and adding it to a document's **LinksLayer**. Delete a link by calling the **Unlink** method, which removes the object from the document as well as disconnecting the link from the ports.

As with any class implementing **IGoLink**, each **GoLink** has two properties, an integer (**UserFlags**) and an object (**UserObject**), for your use. These properties can sometimes be handy to associate some application-specific data with a link without having to define a new class inheriting from **GoLink**.

Link Path

The default link stroke will consist of three segments (four points in the stroke). The end segments, at the ports, will be relatively short. The middle segment will be just a straight line connecting the two short segments at the ports. There is no short end segment if the corresponding port does not have a link port spot (i.e., the value is **NoSpot**). For the short end segments, **GoPort.GetFromLinkPoint** and **GoPort.GetToLinkPoint** give the end points, **GoPort.GetFromLinkDir** and **GoPort.GetToLinkDir** give the directions, and **GoPort.GetEndSegmentLength** gives the lengths.
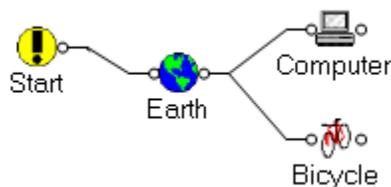

If both ports have link port spots that are **NoSpot**, then the default link stroke consists of only a single segment (two points in the stroke).
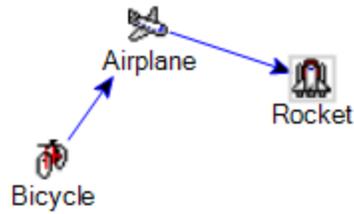
If the link **Style** is **GoStrokeStyle.Bezier**, however, there will be four points in the stroke instead of two, and the curviness is determined by **GoLink.Curviness**. A positive value for this property will result in a clockwise curve; a negative value will result in a counter-clockwise curve.
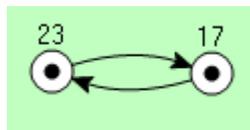
If you set the **Orthogonal** property to **true**, the default link stroke will have five segments instead of three, and all segments will be either horizontal or vertical. When **Orthogonal** is true, setting the **GoStroke.Style** property to **GoStrokeStyle.RoundedLine** will round off the corners of the link. This also helps indicate which direction a particular link is going when several links have co-linear segments.
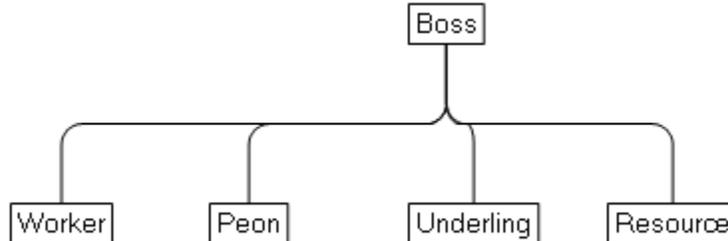
An additional option, when **Orthogonal** is true, is to set the **Style** to **GoStrokeStyle.RoundedLineWithJumpOvers**. This screen shot was taken from the Processor sample application:
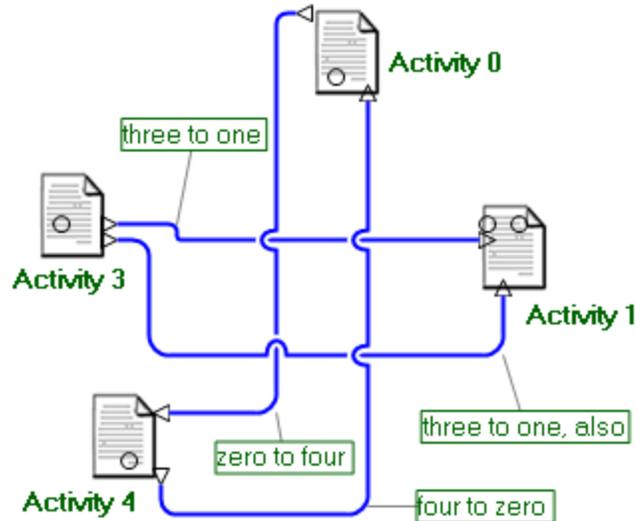
If the position of one or both of its **GoPort**s changes, the **GoLink** redraws itself to connect the new positions. When either port changes it calls the **OnPortChanged** method, which by default just calls **CalculateStroke**. This method is responsible for making sure the stroke goes in the desired manner by having all the right points. Override the **CalculateStroke** method to define your own manner of determining the points used by the link's stroke. For orthogonal links it may be sufficient to override **GetOrthoPoints**, which is called just for adding the two additional midpoints of the default orthogonal link stroke.

When the link's from and to ports are the same port, the default **CalculateStroke** method produces a little "loop" connecting the port with itself.
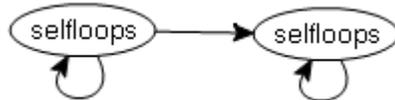
You can control the size of the loop by setting the **Curviness** property; a negative value plots the link on top of the node instead of on the bottom.

Controlling the Link Path

As mentioned above, the points of a **GoLink**'s stroke are determined by the **CalculateStroke** method. **GoLink** provides many different standard paths based on various properties such as **GoLink.Orthogonal** and properties of the ports that the link is connected to, such as **GoPort.FromSpot** and **GoPort.ToSpot**.

You can of course programmatically modify the points of the stroke. The user may also be able to, if it is **Resizable** and **Reshapable**. However, such modifications will be lost as soon as **CalculateStroke** is called again, perhaps due to the repositioning of one of the ports. You can control the overall behavior

of **CalculateStroke** to take any existing points into account by setting the **AdjustingStyle** property. This affects the **AdjustPoints** method, called by **CalculateStroke**, to provide a customized path based on the current points in the stroke.

The **GoLinkAdjustingStyle** enum currently has four defined values:

- **Calculate**, clear the existing points and specify the standard behavior:
    - When both ports have link spots that are **GoObject.NoSpot**, draw a Bezier curve if the stroke style is **GoStrokeStyle.Bezier**, or else draw a straight line.
    - When only one port has a link spot, draw a two-segment stroke with straight lines.
    - When both ports have link spots, draw either a three-segment stroke with straight lines or a Bezier curve
    - If **GoLink.Orthogonal** or **GoLink.IsSelfLoop** is true, draw a five-segment stroke with straight/rounded/jump-over lines (depending on the **Style**) or a Bezier curve

- **Scale**, when there are more than the standard number of points in the stroke, scale and rotate the intermediate points so that the link's shape stays approximately the same. **AdjustPoints** will call the **RescalePoints** method.

- **Stretch**, when there are more than the standard number of points in the stroke, linearly interpolate the intermediate points along the X and Y dimensions between the ports. **AdjustPoints** will call the **StretchPoints** method.

- **End**, when there are more than the standard number of points in the stroke, or if the stroke is orthogonal, just modify the end points, while leaving any intermediate points unchanged. **AdjustPoints** will call the **ModifyEndPoints** method.

Another automatic way of of specifying a stroke path for **Orthogonal** links is to set **GoLink.AvoidsNodes** to true. This actually modifies the behavior of **AddOrthoPoints** to calculate and follow the shortest path between the end points that does not cross over any areas specified as "occupied" by **GoDocument.IsAvoidable** and **GoDocument.GetAvoidableRectangle**.

In order for such path searches to work, the link must first be part of a **GoDocument** so that it can know which nodes to consider avoiding. Thus if you create and connect a link before adding it to a document layer, you will need to explicitly call **GoLink.CalculateStroke** after adding the link to the document.

Appearance and Behavior

Many attributes of links can easily be customized through the properties and methods of **GoStroke** and **GoShape**, such as:

- line color, thickness, and style (**GoShape.Pen** and **GoStroke.Style**)

- arrowheads (**GoStroke** arrowhead properties and **GoShape.Brush**)

- number, location, and size of line segments (**GoStroke** points and **CalculateStroke**)

- number, style, and behavior of resize handles (pick points and **DoResize**)

- highlighting (**GoStroke.Highlight** and **GoStroke.HighlightPen**)

The following image shows two nodes connected by a link. The link is **Orthogonal**, with a **GoStrokeStyle.RoundedLine** style. It has a fuchsia colored dash-dotted pen of width 3, and it has a turquoise highlight pen of width 6. The link has an arrow at the "To" end, and the arrow shaft length is equal to the arrow length to give it a triangular shape. The arrowhead is filled with a forest-green brush. Finally, the link is shadowed.
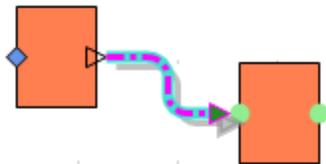
You can customize the appearance of arrowheads by setting arrowhead properties such as **ToArrowLength**, **ToArrowShaftLength**, **ToArrowWidth**, **ToArrowFilled**, **FromArrowLength**, **FromArrowShaftLength**, **FromArrowWidth** and **FromArrowFilled**. Additional customization is possible by overriding **GoStroke** methods and properties.
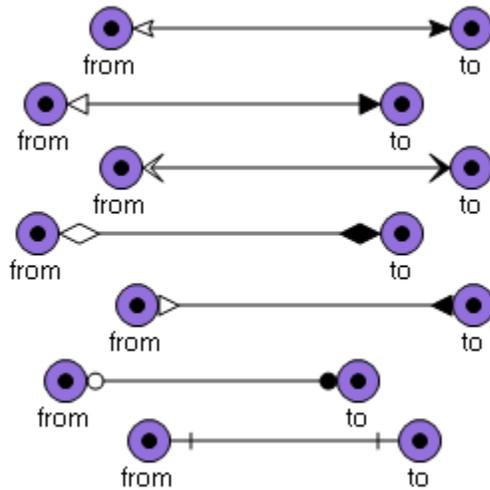
A link's selection handles, like a stroke's, are positioned at the points along the stroke, not along the bounding rectangle. A selected link will not have selection handles at the very end points, unless there are only one or two segments in the stroke.
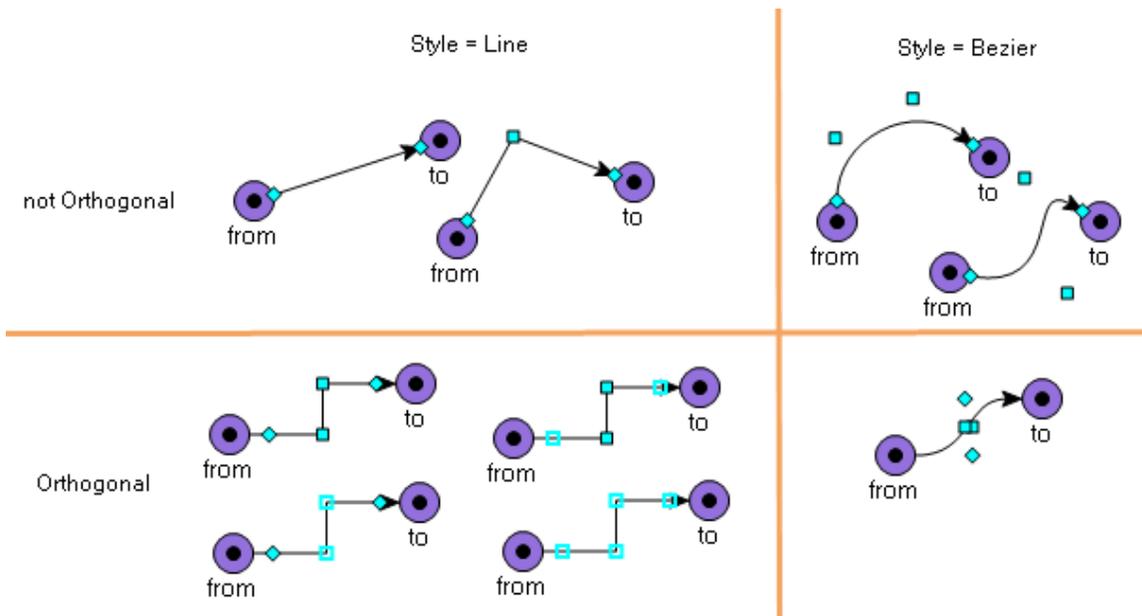
If the link is **Relinkable**, the end selection handles will be diamonds instead of rectangles. Relinking by the user dragging an end selection handle causes the existing link to be disconnected from one port. When the link gesture is completed the port is set again.

Dragging filled rectangular selection handles just moves the stroke point, thus rerouting the link. If the link is orthogonal, the resizing moves that middle

segment to maintain orthogonality. When the link is not **Reshapable**, the rectangular selection handles are hollow, indicating that the user cannot move them.

Movable Links

Normally the ports that a link connects determine the link's position and shape. When one or both ports move, the link moves too. Users should be able to move nodes around, but not links, since that would make the links appear disconnected from their ports. Thus by default **GoObject.Movable** is false for all **GoLink**s.

However, it is possible to implement GoDiagram applications where the user can drag links around, leave them partly connected or completely disconnected, reconnect them by superpositioning a link end with a port by moving either the link or the node, and have nodes automatically drag around their partly connected links. This is demonstrated in the MovableLinkApp sample application by providing a custom dragging tool and by setting **GoLink.Movable** to true. The override of **GoToolDragging.DoDragging** is necessary when the user completes the drag in order to actually set the **GoLink.FromPort** and **GoLink.ToPort**, either to new port values for a new connection or to nothing/null for a disconnection.

Labeled Links

The **GoLabeledLink** class supports up to three additional objects located near either end and near the middle of the link. The **GoLabeledLink** class has three properties: **FromLabel**, **MidLabel**, and **ToLabel**, which can be **nothing**/**null** or any **GoObject.**

The class **GoLabeledLink** does not inherit from **GoLink** but from **GoGroup** instead. The group has up to four children: a **GoLink** and the three labels. **GoLabeledLink** gets its link-ness by implementing **IGoLink**. Most of the link properties and methods are delegated to the child **GoLink**, which is held as the **RealLink** property. So you can change the appearance of a labeled link with code such as:

```
GoLabeledLink l = ...;
Pen p = new Pen(Color.FromArgb(123, 234, 56), 4);
p.DashStyle = DashStyle.DashDot;
l.RealLink.Pen = p;
```

(Remember that you must not modify a **Pen** or a **Brush** after you have assigned it to any **GoObject** or **GoDocument** property.)

OnPortChanged method calls both invoke the child **GoLink**'s **OnPortChanged** method and **LayoutChildren** as well, the latter to maintain the proper positions for the labels.

LayoutChildren just calls the methods **PositionEndLabel** and **PositionMidLabel**, which try to be smart about placing the labels where they do not overlap the link stroke too much, but you can override these methods to implement your own positioning policies. For the relatively common case where you want the object to be centered on the link rather than off to one side, you can just set the **FromLabelCentered**, **MidLabelCentered**, or **ToLabelCentered** properties.

The following image displays two labeled links, each with three labels. The Orthogonal link has the labels at their default positions; the link with the labels ending in "4" have the labels centered along the link's stroke.
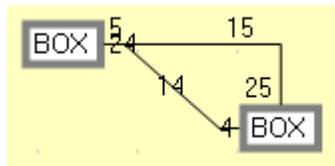
The labels can be any object but are usually instances of **GoText**. One possible use of centered non-text labels is to hold ports, to allow links to come off of links.

If you would like to customize the appearance or behavior of the **RealLink** part of a **GoLabeledLink** by deriving a new class inheriting from **GoLink**, you can get a **GoLabeledLink** to use your custom **GoLink** class by either overriding **GoLabeledLink.CreateRealLink** or just by setting **GoLabeledLink.RealLink** to a new instance of your link class.

```
[Serializable]
public class FancyLink : GoLink {
  public FancyLink () {
    . . . various initializations of GoLink . . .
  }
  . . . various overrides, perhaps . . .
}
[Serializable]
public class MyLabeledLink : GoLabeledLink {
  public MyLabeledLink () {
    . . . various initializations of GoLabeledLink . . .
  }

  public override GoLink CreateRealLink() {
    return new FancyLink();
  }
}
```

GoLink has a **GoLink.AbstractLink** property that will return the **GoLabeledLink** if the **GoLink** is part of a **GoLabeledLink**; otherwise it will just return itself.

5. VIEWS AND TOOLS

GoView is a **Control** that supports the display and editing of diagrams containing graphical objects such as nodes and links.

GoView supports the model-view-controller architecture. **GoDocument** is the model for **GoView**.

GoView supports many basic features:

- displaying a **GoDocument** and its **GoLayer**s of **GoObject**s

- displaying its own view-specific layers of objects, such as selection handles

- painting a background and optionally drawing a grid

- borders

- optional Controls along all four sides and at all four corners

- scrolling [in Windows Forms, scroll bars and autoscrolling; in Web Forms, scroll margins/buttons]

- panning, when the user clicks on the mouse wheel [automatic in Windows Forms]

- scaling (zooming)

- printing [GoDiagram Win only]

- generating a bitmap for part or all of the document

- selection

- clipboard transfer: cut, copy, and paste

- drag-and-drop, both within a window as well as between windows [latter is GoDiagram Win only]

- view events such as **ObjectSingleClicked, BackgroundDoubleClicked**, **ObjectGotSelection**, **ObjectLostSelection**, **ObjectEnterLeave, SelectionDeleted, BackgroundSelectionDropped, ClipboardPasted**

- in-place text editing and other controls [Windows Forms only]

- tooltips for objects

- cursors for objects

- default cursor for view [GoDiagram Win only]

- passing unified input events to the current **GoTool**

- properties to enable or disable selecting, moving, copying, resizing, deleting, inserting, linking, editing, mouse input, keyboard input, dragging out

- drop shadows

- greeking

Views have a number of **GoTool** instances that they use to handle mouse and keyboard input. The following predefined tools are typically used:

- **GoToolManager** – selection, choosing other tools to run, default keyboard commands

- **GoToolAction** – support for individual objects such as buttons or knobs that need to get mouse down, mouse move, and mouse up events

- **GoToolContext** – context menu support for objects (but context menus are only available on Windows Forms)

- **GoToolCreating** – construction and automatic resizing of new objects

- **GoToolDragging --** moving and copying objects

- **GoToolLinkingNew --** drawing new links between ports

- **GoToolRelinking --** reconnecting existing links to different ports

- **GoToolResizing --** resizing objects

- **GoToolRubberBanding** – rubber-band box selection

- **GoToolSelecting** – may change the selection on mouse up when no other tool is invoked

- **GoToolZooming** – rubber-band specification of the view's new document position and scale

- **GoToolPanning** – automatic panning controlled by the direction and distance the mouse is from an initial point (also supports manual panning, separately)

Display

The primary purpose of **GoView** is to display a **GoDocument** and its **GoObject**s. You can use the default **GoDocument** that the view creates, or you can supply your own by setting the **Document** property. It is also common to override **CreateDocument** so that the constructor for your view subclass will automatically create your own document class too.

A **GoView** is just a regular **Control**. The part of a **GoView** that shows the document is called the canvas. A view can also have scroll bars in Windows Forms or a scrolling margin in Web Forms. A view, like any control, can have a border surrounding it, but the canvas area by itself does not support one.

GoView also supports the display of its own view-specific objects. Thus each view on the same document can have its own set of **GoObject**s. These view objects will appear in front of all document objects. The most common example of a view object is a selection handle (a **GoHandle**).

Scrolling

GoView has built-in support for scrolling and either scroll bars (in Windows Forms) or scroll buttons in a scrolling margin (in Web Forms).

Because a view does not necessarily show the whole document, the **DocPosition** property indicates where the view's top-left corner is in the document. The **DocExtentSize** property indicates the size of the view's canvas in the document.

Each view also provides the **DocumentSize** and **DocumentTopLeft** properties, which allow each view to have a potentially different notion of the document it is looking at. In particular, the **ShowsNegativeCoordinates** property affects the value of both of these properties. A **true** value allows the user to see objects positioned anywhere in the document. The value of the **DocumentSize** property is then the same as the value of **Document.Size** and the value of the **DocumentTopLeft** property is the same as the value of **Document.TopLeft**. This can be convenient when additional objects need to be added to the left of the existing ones, and you don't want to shift the existing ones rightwards to avoid negative coordinates. A **false** value prevents users from scrolling to parts of the document at negative coordinates. The **DocumentSize** and **DocumentTopLeft** properties are changed to pretend the document only has coordinates at non-negative positions.

Scrolling in Windows Forms

For Windows Forms, there will be both a horizontal and a vertical scroll bar, but you can remove one or both of them by setting the respective properties to **nothing**/**null**. There is also a separate corner component, where the two scroll bars meet, that is visible when both scroll bars are visible.
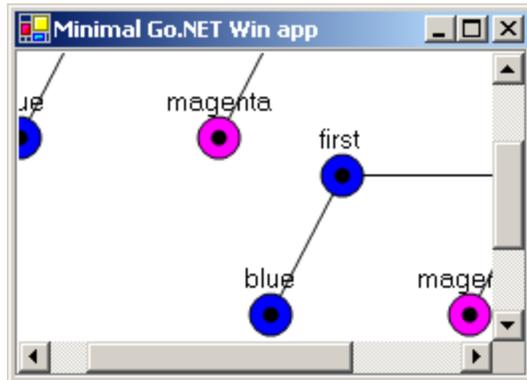
The **ShowVerticalScrollBar** and **ShowHorizontalScrollBar** properties control when the scroll bars are visible. The default value of **GoViewScrollBarVisibility.IfNeeded** will result in the scroll bar being visible only when the view is too small to display the whole document in the respective direction.

Of course users can scroll the view by manipulating the scroll bars. The **ScrollSmallChange** property determines how much the view scrolls when the user clicks on an arrow in a scroll bar.

Programmatically you can call the **ScrollPage** and **ScrollLine** methods to scroll by most-of-the-window and by **ScrollSmallChange** amounts. The standard implementations of Page-Up, Page-Down and mouse wheel turns call these methods.

When drag-and-drop is enabled, users can also cause automatic scrolling when they are dragging near the edge of the canvas. This autoscroll region is specified by the **AutoScrollMargin** property. You can disable this behavior by setting the margin width and height to zero. You can customize how long to wait in the autoscroll region by changing the **AutoScrollDelay** property, and how quickly it scrolls by changing the **AutoScrollTime** property.

Scrolling in Web Forms

For ASP.NET Web Forms, by default there is a scrolling margin that is drawn around the image. Small arrows are drawn and enabled for each direction in which the view may be scrolled.
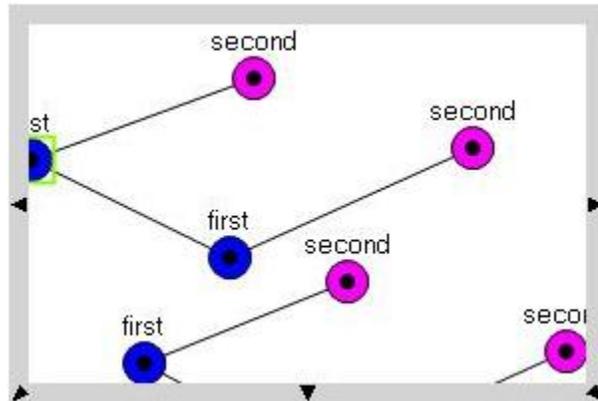
The size and appearance of the scrolling margin may be customized by setting the **ScrollMarginBackgroundColor**, **ScrollMarginForegroundColor**, and **ScrollMarginSize** properties. You can control whether any scroll buttons are drawn for a particular orientation by setting the **ShowsVerticalScrollButtons** and **ShowsHorizontalScrollButtons** properties.

When the user clicks in the margin, the view's **ScrollLine** method is called appropriately to shift the **DocPosition** by the **ScrollSmallChange** amount. A Shift-click will result in a call to **ScrollPage** instead.

Scaling and Coordinate Systems

GoView also supports zooming, to change the scale at which the objects are drawn. The **DocScale** property is normally 1.0f; smaller values make objects appear smaller on the screen; larger values correspond to zooming into the diagram. For example, when the **DocScale** value is 0.5f, objects will appear half as large as normal.

When setting the **DocScale** property the **GoView.LimitDocScale** method is called to ensure a new value for the **DocScale** property meets your requirements—by default it makes sure the scale is between 0.01f and 10.0f. If you want to extend or modify the permitted range, perhaps even computed dynamically, you will need to override **GoView.LimitDocScale**.

The ability to scroll and zoom the view means that the coordinate system used in a view is different from that used in the document. The overloaded **ConvertDocToView** and **ConvertViewToDoc** methods perform the basic transformations between document **PointF**s, **SizeF**s, and **RectangleF**s and view **Point**s, **Size**s, and **Rectangle**s.

The **RescaleToFit** method changes the **DocScale** property so that all of the objects in the document can be seen in the view without scrolling.

The **RescaleWithCenter** method changes the **DocScale** and tries to keep the view centered about a given document point.

Painting

As a control, **GoView** overrides **OnPaint** in order to render the view. This is responsible for scaling and translating the **Graphics** and getting a document-coordinates clipping rectangle. It then calls **PaintView**, which calls methods to fill in the paper color (**PaintPaperColor**), to draw any additional background such as an image (**PaintBackgroundDecoration**), and then to draw all of the layers of document objects and view objects (**PaintObjects**), including any grid or sheet of paper that may be held in the **BackgroundLayer** of the view.

You can override **PaintView** or any of the three methods called by **PaintView** in order to get different effects; overriding **PaintPaperColor** and **PaintBackgroundDecoration** are the most common. **PaintPaperColor** uses the **Control.BackColor** property when the view's document's **PaperColor** property is not **Color.Empty**. The **Control.ForeColor**, **Control.Text**, and **Control.Font** properties are currently not used.

The **PaintView** method uses the **SmoothingMode**, **TextRenderingHint**, and **InterpolationMode** properties to control the quality of how all objects are painted. If you want to change how a particular kind of object is drawn, for example if you want lines to be drawn with jagged edges rather than smoothly with anti-aliasing, you will need to override the **GoObject.Paint** method for that object.

The **GetBitmapFromCollection** method returns a **Bitmap** holding the rendering of all of the objects in the argument collection. The bitmap does not include any background or view objects.

The **GetBitmap** method returns a **Bitmap** of the view itself, at the current **DocScale** and **DocPosition**, with the current background and all visible document and view objects in the **DisplayRectangle**.

Views support the notion of *greeking*, which simplifies or omits the painting of objects at small scales. This effect helps avoid clutter and improves performance, particularly when the painted area would be too small for the user to see well. The **PaintGreekScale** and **PaintNothingScale** specify the default scales at which a simplified rendering and at which no rendering should occur. Normally only the **GoText** and **GoPort** classes perform greeking.

Printing

GoView also provides support for printing in GoDiagram Win applications. The **Print** method brings up the print dialog and then starts a **PrintDocument**, which repeatedly calls **PrintDocumentPage**. You can easily override **PrintDocumentSize**, **PrintDocumentTopLeft**, and **PrintScale** to customize how much is printed, on how much of the page, and at what scale. **PrintScale** can also be set. Override **PrintDecoration** to add headers and/or footers or any other decoration on each page. Override **PrintView**, like **PaintView**, to change what things get printed--by default the paper color and the view objects are not

printed. But you can set **GoView.PrintsViewObjects** to true to show view objects such as selection handles.

Of course there is no particular support for printing in Compact Framework or Web Forms applications. However, for GoDiagram Web, we have added a new **WebControl**, **GoPrintView**, which may help you implement ASPX pages so that the user can print multipage diagrams.

Selection

Each **GoView** has a **GoSelection** that holds the currently selected document objects for that view. The default selection object is an instance of **GoSelection**, but you can override **GoView.CreateSelection** to return your own subclass. The selection object is also responsible for managing selection handles in the view. Many events and methods in **GoView** deal with the current selection, either by changing it, or by operating on its collection of objects. Important examples include: **EditCut**, **EditCopy**, **EditPaste**, **DeleteSelection**, **MoveSelection**, **CopySelection**, **SelectAll**, **SelectInRectangle**, and **SelectNextNode**.

GoSelection implements **IGoCollection**, so you can use the **Add**, **AddRange**, **Remove**, **Contains**, and other collection methods for programmatically manipulating the selection. **GoSelection** has additional methods such as **Select**, which makes its argument the one and only selected object, and **Toggle**, which **Add**s the argument if it wasn't in the selection or **Remove**s it if it was.

As with any .NET collection, you can easily iterate over the objects in the selection by using the **foreach** construct. But it is important to remember that you must not modify the selection while you are iterating, if you want to avoid unpredictable behavior. It is very easy to make this mistake accidentally. Perhaps the most commonly programmed error is to iterate over the selection, removing the objects from the document along the way. But removing an object from a document will also have the side effect of removing it from the selection of each view of the document.

The first selected object is known as the primary selection; any other selected objects form the secondary selection. **Primary** is a read-only property whose value is the primary selection, or nothing/null if no object is selected. You can restrict the number of selected objects for a view, the **Count** property, by setting the **GoView.MaximumSelectionCount** property.

GoSelection is also responsible for creating handles for selected objects. The **CreateBoundingHandle** and **CreateResizeHandle** methods are responsible for allocating handles of the appropriate size and position, associating them with the selected object, and then adding them to the view. **RemoveHandles** is responsible for disassociating them with the selected object and removing them from the view. You can look for an existing handle for a particular object (in a particular view/selection) by using **FindHandleByID**. You can iterate over all the handles for a selected object by using **GetHandleEnumerable**.

The **GoView.ResizeHandleSize** and **ResizeHandlePenWidth** properties determine the default size and appearance for resize handles. **GoView.BoundingHandlePenWidth** help determine the appearance of bounding handle rectangles. Handle IDs help distinguish between multiple handles for the same selected object.

The **GoView.PrimarySelectionColor** and **SecondarySelectionColor** control the color of selection handles. When the view loses focus, the **NoFocusSelectionColor** is used instead, unless the **HideSelection** property is true, in which case the selection handles all disappear when the view does not have focus.

For most applications, a user will expect that the top-level nodes (and links) of a diagram are what the user can select. Thus these objects, which are usually instances of subclasses of **GoGroup**, will be the objects in the **GoSelection** collection.

However, it is often the case that what gets a selection handle is not the top-level object, a group, but some child object inside the group. For example, users may expect to select and resize the icon of a node. Thus resize handles should not be on the whole group, but just on the node's icon. Similarly, a rubber band selection rectangle need not include all of a node to select it, but just the node's entire icon.

To enable this sleight-of-hand, **GoObject** has a **SelectionObject** property that defaults to **this** object itself. A class implementing the above example node would override **SelectionObject** as follows:

VB.NET:
```vb
Public Overrides ReadOnly Property SelectionObject() As GoObject
  Get
    If Not Me.Icon Is Nothing Then
      Return Me.Icon
    Else
      Return Me
    End If
  End Get
End Property
```

C#:
```csharp
public override GoObject SelectionObject {
  get {
    if (this.Icon != null)
      return this.Icon;
    else
      return this;
  }
}
```

Then when an object gets selected or loses it, it calls **AddSelectionHandles** or **RemoveSelectionHandles** not on itself but on its **SelectionObject**, here its icon. The distinction between the two objects is carried on by the **IGoHandle** interface—typically the **IGoHandle.SelectedObject** property refers to the top-level node; the **IGoHandle.HandledObject** property refers to the top-level node's **SelectionObject**.

Grids

Each view can display a grid, using a **GoGrid** that is available as the **GoView.BackgroundGrid** property and that is held in the **BackgroundLayer** of the view. The grid's properties are accessible either directly through the **GoView.BackgroundGrid** property or via the **GoView.Grid…** properties. The grid is not part of a document, so that not all views on a document have to display a grid or the same grid. But the spacing and sizing of the grid, like view objects, are measured using document coordinates.
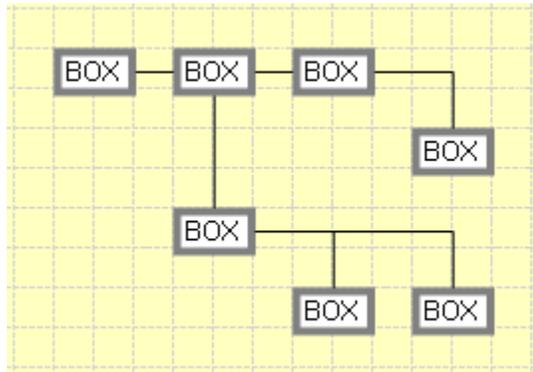
The **GridStyle** property specifies whether the grid is drawn as dots, crosses or lines, the latter either in both directions or just horizontally or just vertically.

The **GridOrigin** and **GridCellSize** properties control the spacing of the grid's cells and whether the grid starts at (0, 0). The cell size is independent of the distance the scroll bar scrolls when the user clicks on a scroll bar arrow or a scroll button.

The **GridLineColor**, **GridLineWidth**, **GridLineDashStyle**, and **GridLineDashPattern** properties all control how the grid lines are drawn.

You can display both major lines and minor lines by specifying the **GridMajorLineFrequency** property. Positive values indicate how often vertical and horizontal lines should be drawn as "major" lines. Just as the **GridLine…** properties control the appearance of regular (or "minor") lines, the **GridMajorLine…** properties control the appearance of major lines.

The **GridSnapDrag** property controls whether a user's dragging of objects automatically relocates them to the grid points.

The **GridSnapResize** property controls whether a user's resizing of an object automatically positions its bounds to the grid.

Grid snapping moves the **Location** of objects (when dragging) to a point in a grid cell. You can control which spot in the cell that is by setting the **GridSnapCellSpot** property.

Sheets

Each view can also display what appears to be a sheet of paper. This is implemented by having a **GoSheet** that is held in the **BackgroundLayer** of the view, accessible via the **GoView.Sheet** property. Initially there is no sheet, but by setting **GoView.BackgroundHasSheet** property to true, one will be created by calling **GoView.CreateSheet** and setting **GoView.Sheet**.

When there is a **GoView.Sheet**, all of the **GoView.Grid…** properties refer to the **GoView.Sheet.Grid**, rather than to the **GoView.BackgroundGrid**. By default the sheet's grid is limited to the sheet of paper.
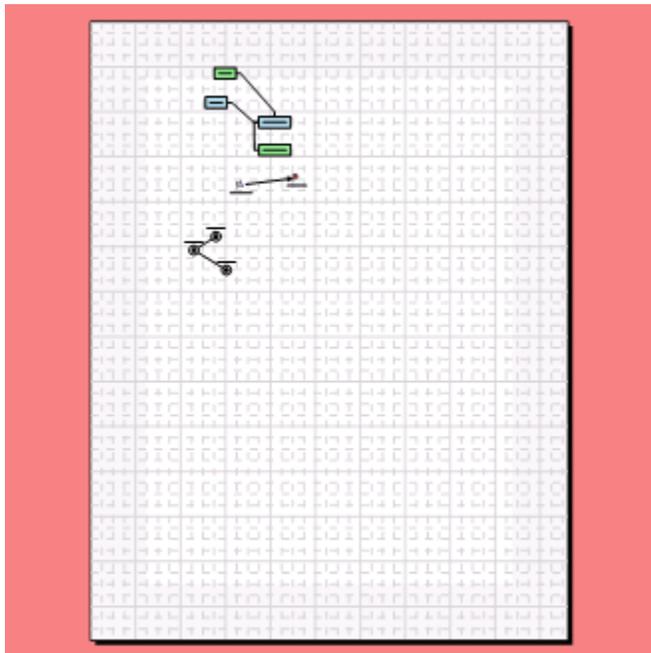
The above screen shot shows a **GoView** with **BackgroundHasSheet** set to true. **GoView.SheetStyle** is set to **GoViewSheetStyle.Sheet**, so that the view's **GoSheet** is visible. The **Control.BackColor** is set to **Color.LightCoral**; the **GoDocument.PaperColor** is **Color.White**. The **GoView.BackgroundGrid**, which covers the whole **GoView**, is not visible and is not used. The **GoView.Sheet.Grid** displays both major and minor grid lines.

Each sheet can also show the paper margins. In the screenshot above it is barely visible as a very translucent gray drawn along the edges of the sheet. The

SheetShowsMargins, **SheetMarginColor**, **SheetTopLeftMargin**, and **SheetBottomRightMargin** properties control the size and appearance of any margins. The sizes of the margins must be set explicitly by your application if you want them to reflect the size of any print pages, since each **GoView** does not know about any printers that the user may have chosen.

How much of the sheet is shown in the view as the view is resized is controlled by the **SheetStyle** and **SheetRoom** properties, and is implemented by the **GoView.UpdateExtent** method. A **GoView.SheetStyle** value of **GoViewSheetStyle.WholeSheet**, for example, will automatically rescale and scroll the view as the view changes size, so that the whole sheet remains visible, much as in the screen shot above. A value of **GoViewSheetStyle.Sheet** will have the sheet be visible, but the view will not automatically rescale and scroll as its size is changed. The default value is **GoViewSheetStyle.None**—the **Sheet** is not visible and **UpdateExtent** does nothing. The value of **GoView.SheetStyle** does not limit the user's scrolling and/or zooming.

When there is a **Sheet** and the **SheetStyle** is not **None**, printing is limited to a single sheet of paper. If you really want to print multiple pages, you can temporarily set the **SheetStyle** to **None**, or you can override the **PrintDocumentSize**, **PrintDocumentTopLeft**, and **PrintScale** properties to calculate the values you need.

Shadows

Each view has a notion of the standard shadow to be used for objects that display a drop-shadow effect and have a true value for the **GoObject.Shadowed** property.

The effect is controlled by the following **GoView** properties and methods: **ShadowOffset**, **ShadowColor**, **GetShadowBrush**, and **GetShadowPen**. Each object can change the standard appearance by overriding **GoObject.GetShadowOffset**, **GoObject.GetShadowBrush**, and **GoObject.GetShadowPen**.

Events

GoView is responsible for handling events that occur when the user interacts with the view. Each **GoView** also has view-specific state that other code may care about tracking. Because **GoView** is a **Control**, most cases are handled by the predefined **Control** events. In fact, all of the additional properties that **GoView** defines are covered by the **GoView.PropertyChanged** event.

However **GoView** does define additional events that more abstractly deal with common user actions. These events are:

- **ObjectSingleClicked** – the user clicked on an object

- **ObjectDoubleClicked** – the user clicked quickly twice on an object

- **ObjectContextClicked** – the user context clicked on an object

- **ObjectSelectionDropReject** – the user is dragging the selection on an object—allow an event handler or the object to reject a drop

- **ObjectSelectionDropped** – the user dropped the selection on an object

- **ObjectHover** [GoDiagram Win only] – the user has left the mouse motionless over an object for a while determined by **GoView.HoverDelay**

- **ObjectEnterLeave** – the user has moved the mouse into or out of a document object, either as a mouse-over or as a dragging of the selection

- **BackgroundSingleClicked** – the user clicked in the background

- **BackgroundDoubleClicked** – the user double clicked in the background

- **BackgroundContextClicked** – the user context clicked in the background

- **BackgroundSelectionDropReject** – the user is dragging the selection in the background—allow an event handler to reject a drop

- **BackgroundSelectionDropped** – the user dropped the selection in the background

- **BackgroundHover** [GoDiagram Win only] – the user has left the mouse motionless in the background for a while determined by **GoView.HoverDelay**

- **ObjectGotSelection** – an object has been added to the current selection

- **ObjectLostSelection** -- an object has been removed from the current selection

- **SelectionStarting** – some operations that may make many changes to the view's **Selection** will surround all of the **ObjectGotSelection** and/or **ObjectLostSelection** events with a **SelectionStarting** event beforehand and a **SelectionFinished** event afterwards. If you have other Controls that you want to keep up-to-date with the **Selection**, you can optimize updating those Controls using these two paired events.

- **SelectionFinished** – (see **SelectionStarting**)

- **SelectionMoved** – the user finished moving the selected objects

- **SelectionCopied** – the user just copied the selected objects

- **SelectionDeleting** – the user is about to delete the selected objects; the deletion can be cancelled

- **SelectionDeleted** – the user has just deleted the selected objects

- **LinkCreated** – the user finished drawing a new link

- **LinkRelinked** – the user finished reconnecting an existing link

- **ObjectResized** – the user finished resizing an object

- **ObjectEdited** [Windows Forms only] – the user finished editing an object

- **ClipboardPasted** – the user just pasted something from the clipboard

Some of these events have no corresponding document change. Others, such as **SelectionMoved**, clearly involve changes to objects in a document. The difference is that the **SelectionMoved** event is specific to a view and only occurs after the user has moved the selected objects. There are **GoDocument.Changed** events for each object that gets moved even if the move occurs programmatically rather than interactively. For some events like **GoView.SelectionMoved**, there is a further difference in that this event only occurs once, even if **GoView.DragsRealtime** is true, but there will be many **GoDocument.Changed** events for the multiple objects moved many times during a drag.

As with other **Control** events, you can add your own event handlers for these events, or if you have your own subclass of **GoView**, you can override the **On…** methods to handle the events. These events are described more fully later in this chapter.

Changes to objects that belong to a view, including insertions and removals, provide notification through the **RaiseChanged** method, just as for documents. However, there is no **GoView.Changed** event for **GoObject** changes and thus no **GoView.OnChanged** method. Getting notification of changes to view objects is rarely needed. But if it is necessary, you can override **RaiseChanged** to observe changes to view objects.

Document Changed Events and Views

GoView handles **GoDocument.Changed** events, which is how it can keep its display up-to-date with changes to the document and its objects. The method **GoView.OnDocumentChanged** is invoked to handle document changes. This method notices when objects are inserted, changed, or removed, or when other document or layer changes occur that affect the display in the view. It then invalidates the appropriate regions, so that the **OnPaint** method is called at a later time to actually repaint the objects visible in those regions.

You can override **GoView.OnDocumentChanged** if you want your own view-specific code to respond to changes to documents or document objects. This is preferable to adding event handlers to a document if you are defining your own subclass of **GoView**.

Input Events

GoView provides a slightly more general notion of mouse and keyboard input by using the **GoInputEventArgs** class. This class holds unified input event args information. It holds the position where the mouse event occurred, in both view and document coordinates. It also remembers the mouse buttons, such as **MouseButtons.Right**, and key modifiers, such as **Keys.Control**. For keyboard input, it holds the key that was pressed, along with the key modifiers. For drag-and-drop events, the information is like that for mouse events. Mouse wheel rotation events are included too, with the **Delta** property.

In case you need additional information, the original Windows Forms **MouseEventArgs**, **DragEventArgs**, or **KeyEventArgs** is kept in the **GoInputEventArgs** too. For Web Forms, of course, these events are only simulated, and the Compact Framework does not support **DragEventArgs**.

GoView overrides the low-level **OnKeyDown**, **OnMouseDown**, **OnMouseMove**, **OnMouseUp**, **OnDoubleClick**, **OnMouseWheel**, **OnDragOver**, and **OnDragDrop** methods to capture the input event information. This information is remembered in a **GoInputEventArgs** instance as the **GoView.LastInput** property.

For the convenience of code that needs to remember the input state at the time of a mouse down, **OnMouseDown** also remembers the input event information in a separate **GoInputEventArgs** instance as the **FirstInput** property.

If you do nothing to override the input handling of a **GoView**, the default behavior gives you input handling that anyone familiar with a graphical object editor would expect. Objects can be selected, moved, and resized using the left mouse button. Multiple selections can be made using shift-left button or control-left button or with rubber-band selection. Links can be created by left button down and drag on a **GoPort**.

Tools

The code implementing the standard view behaviors is not actually in the **GoView** class. Instead, input events are passed on to instances of small, narrowly defined classes that are responsible for implementing the policies and mechanisms of user input. These classes implement **IGoTool**, and are normally inherited from **GoTool**.

A normal instance of **GoView** will have a set of tools that it can use. Each **GoView** always has a current tool, held by the **Tool** property. Each **GoView** also has a **DefaultTool** property. The initial tool is also the default tool, which is

created as a result of the view's constructor calling **CreateDefaultTool**. The currently selected tool implements the view's "mode" of user interaction.

GoView starts each tool by calling its **Start** method, to give it a chance to initialize any state. As each tool runs, it handles unified input events with the **DoMouseDown**, **DoMouseMove**, **DoMouseUp**, **DoMouseHover**, **DoCancelMouse**, **DoMouseWheel**, and **DoKeyDown** methods. When the view's current tool is set to a new tool, **GoView** calls the old tool's **Stop** method so that it can clean up before starting the new tool. A tool can terminate itself by calling **GoTool.StopTool**, which sets the view's current tool to null. Setting **GoView.Tool** to nothing/null stops the current tool, sets the **Tool** property to the value of the **DefaultTool** property, and starts that default tool.

GoToolManager and Standard GoView Tools

The normal default tool is an instance of **GoToolManager**. This tool's primary purpose is to implement the default keyboard commands and to invoke "mode-less" tools according to the object (if any) at the mouse point.

GoView divides up its set of tools into three lists, one each for mouse down and mouse move and mouse up, according to when the particular tool is likely to be startable. **GoToolManager**'s **DoMouseDown**, **DoMouseMove**, and **DoMouseUp** methods then just iterate through the corresponding list of tools to find the first one whose **CanStart** method returns true. The **CanStart** method is responsible for looking at the current state of the document, the view and the current input event and deciding if it is appropriate for that tool to start operating. As soon as the tool manager finds such a tool, it makes that tool the view's current tool, thereby stopping itself and starting the selected tool.

The **GoView.MouseDownTools** list normally includes instances of **GoToolAction**, **GoToolContext**, **GoToolRelinking**, and **GoToolResizing**. These tools expect to get **DoMouseMove** and **DoMouseUp** calls during their operation.

The **GoView.MouseMoveTools** list normally includes instances of **GoToolLinkingNew**, **GoToolDragging** and **GoToolRubberBanding**. These tools expect to get additional **DoMouseMove** and **DoMouseUp** calls during their operation.

The **GoView.MouseUpTools** list normally normally contains only an instance of **GoToolSelecting**. This tool does not expect to get any additional **DoMouseUp** calls during its operation, because starting the tool also stops it.

Here's a class hierarchy diagram, starting with **GoTool**:
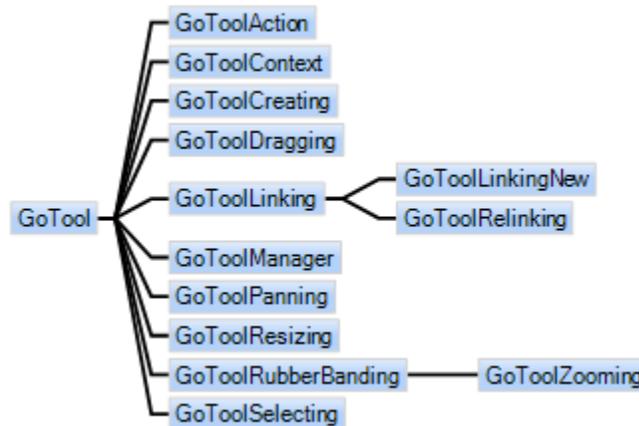
The low-level event handlers capture input event information and then call view methods to perform the default action, which is to invoke the current tool's corresponding method. Finally they call the base methods, so that the respective event handlers are all called. For example, **OnMouseMove** calls **DoMouseMove** followed by **base.OnMouseMove**. **DoMouseMove** just calls **this.Tool.DoMouseMove**. The reason for this indirection is to allow you to put your event handling code in either the view, the tool, or the object, whichever is most sensible for organizing your program.

Many events are ignored in Windows Forms if the view does not have focus. A mouse down event will try to acquire focus for the view.

The Demo1 sample includes a custom tool to allow users to create **GoStroke** instances by clicking with the mouse where the points of the stroke should be. This tool is used in a "modal" fashion, so it is not included in the lists of mouse tools whose **CanStart** methods are called. A command can enter this stroke-drawing mode just by setting the view's **Tool** property to an instance of this tool.

The **GraphView** class, also in the Demo1 sample, uses two customized link-drawing tools to highlight ports during linking and during relinking. The constructor has the code that replaces the standard linking tools, so that only the newly modified tools are used when the user performs a linking action in a view.

High Level Mouse Events

Many of the events defined for **GoView** are more abstract than mouse or key actions. Examples include **SelectionMoved**, **LinkCreated** and **ObjectEdited**, although they are all instigated by the user's mouse or key actions.

For selection changes, the **ObjectGotSelection** and **ObjectLostSelection** events notify all registered **GoSelectionEventHandler**s that an object has just been selected or deselected. The **GoSelectionEventArgs** class has a **GoObject** property that indicates the object in the document.

The **SelectionMoved** and **SelectionCopied** events are raised by the **GoToolDragging** tool after the user has moved or copied the currently selected objects. Unlike **GoDocument.Changed** events on the individually selected objects or mouse move events on the view, the **SelectionMoved** and **SelectionCopied** events only occur once the user has successfully moved or copied the selection.

The **SelectionDeleting** and **SelectionDeleted** events occur just before and after the user is deleting the currently selected objects, in the implementation of **GoView.DeleteSelection**. **SelectionDeleting** has a **CancelEventArgs**, which allows code to stop the deletion by setting the **Cancel** property to true.

The **LinkCreated** and **LinkRelinked** events are raised by the **GoToolLinkingNew** and **GoToolRelinking** tools, respectively, when the user has successfully completed those operations.

The **ObjectResized** event is raised by the **GoToolResizing** tool after the user's resizing is complete. Again, the event happens only once, whereas a **GoObject.ChangedBounds** subhint **Changed** event may occur repeatedly as the user is resizing the object, particularly if **GoObject.ResizesRealtime** is true.

The **ObjectEdited** event should be raised by those objects implementing **GoObject.DoEndEdit**, such as **GoText** for in-place editing. This event only applies to Windows Forms.

The **ObjectSelectionDropReject** event is raised during a drag-and-drop to allow any **GoObject** the opportunity to reject a drop. Similarly, the **BackgroundSelectionDropReject** event is raised during a drag-and-drop when the pointer is not over any document object. To reject the drop, you can set:

```
e.InputState = GoInputState.Cancel
```

When the **…SelectionDropReject** event is not cancelled, the corresponding event occurs for the drop: **ObjectSelectionDropped** or **BackgroundSelectionDropped**.

The **ExternalObjectsDropped** event occurs in **GoView.DoExternalDrop**, so that you can get notification when **GoObject**s are copied into the view due to a drag-and-drop that started from another window. This is convenient when you want to modify the dropped objects, which will be selected, perhaps to move them or to change some of their properties. Note that if you define your own data formats to be handled on a drop, such as **String**s causing particular nodes to be created and added to your document, then the **ExternalObjectsDropped** event is not raised.

The **ClipboardPasted** event occurs during **GoView.EditPaste**, not during **GoView.PasteFromClipboard** or **GoDocument.CopyFromCollection**, which are more general methods for copying objects.

Mouse Click Events

One of the fundamental functions of **GoView** is the ability to handle mouse clicks. The selection may change or a click may be passed on to any visible object on top at that point. This will cause the view to raise events for the benefit of any interested handlers.

For clicks, the event depends on whether there was a selectable object at the mouse point and what kind of click it was. The **ObjectSingleClicked**, **ObjectDoubleClicked**, **ObjectContextClicked**, and **ObjectHover** events notify all registered **GoObjectEventHandler**s that an object was clicked in a certain manner, or that the mouse rested for a while at one spot over an object. The **GoObjectEventArgs** type has a **GoObject** property to indicate the object, and because this class inherits from **GoInputEventArgs**, the event position, buttons, and modifiers are available also.

When there is no object at the click point, the **BackgroundSingleClicked**, **BackgroundDoubleClicked**, **BackgroundContextClicked** and **BackgroundHover** events notify **GoInputEventHandler**s. The event args type is **GoInputEventArgs**, which provides the mouse event information, but of course there is no **GoObject** associated with this event.

GoView does not affect the behavior of the **Control.Click** event. You can use it in the unlikely case that you don't care where the user clicks in the view. Similarly, **GoView** does not affect the behavior of the **Control.MouseHover** event in GoDiagram Win, because that event only happens at most once while the mouse stays inside the view, even if the user moves the mouse and stays over different objects.

A single or double click will invoke either **DoSingleClick** or **DoDoubleClick**. A logical right mouse click will invoke the **DoContextClick** method. These methods all behave similarly. They each try to find the selectable object underneath the mouse event point. If they find nothing, they raise the appropriate background clicked event.

If they do find an object, they first raise the appropriate object clicked event. Then they call the object's **On…Click** method, such as **OnSingleClick**. This gives the object a chance to implement click behavior in its defining class, rather than by overriding methods in a view or by adding view event handlers.

If the **GoObject.On…Click** method returns false, the object's parent group's **On…Click** method is called, on up the parent tree, until the **On…Click** method returns true or until there is no parent. This behavior allows a group to define default behavior for all of its parts; e.g. when a particular part does not handle the click by returning true from the **On…Click** method.

Context Menus

To implement context menus customized for your nodes, you should override **GoObject.GetContextMenu** or **GetContextMenuStrip** in your node class. For example:

VB.NET:
```
Public Overrides Function GetContextMenu(ByVal v As GoView) As GoContextMenu
  If (TypeOf v Is GoOverview) Then Return Nothing
  Dim cm As ContextMenu = New GoContextMenu(v)
  If (CanDelete()) Then
    cm.MenuItems.Add(New MenuItem("Cut",
                          New EventHandler(AddressOf Cut_Command)))
  End If
  If (CanCopy()) Then
    cm.MenuItems.Add(New MenuItem("Copy",
                          New EventHandler(AddressOf Copy_Command)))
  End If
  If (cm.MenuItems.Count > 0) Then
    cm.MenuItems.Add(New MenuItem("-"))
  End If
  cm.MenuItems.Add(New MenuItem("Properties",
                    New EventHandler(AddressOf Properties_Command)))
  Return cm
End Function

Public Sub Cut_Command(ByVal sender As Object, ByVal e As EventArgs)
  If TypeOf sender Is MenuItem Then
    Dim v As GoView = GoContextMenu.FindView(CType(sender, MenuItem))
    v.EditCut()
  End If
End Sub
```

C#:
```
public override GoContextMenu GetContextMenu(GoView v) {
  if (v is GoOverview) return null;
  ContextMenu cm = new GoContextMenu(v);
  if (CanDelete())
    cm.MenuItems.Add(new MenuItem("Cut",
                          new EventHandler(this.Cut_Command)));
  if (CanCopy())
    cm.MenuItems.Add(new MenuItem("Copy",
                          new EventHandler(this.Copy_Command)));
  if (cm.MenuItems.Count > 0)
    cm.MenuItems.Add(new MenuItem("-"));
  cm.MenuItems.Add(new MenuItem("Properties",
                    new EventHandler(this.Properties_Command)));
  return cm;
}
```

```
public void Cut_Command(Object sender, EventArgs e) {
  GoView v = GoContextMenu.FindView(sender as MenuItem);
  if (v != null)
    v.EditCut();
}
```

Note the use of **GoContextMenu**, which is implemented for both Windows Forms and Web Forms. **GoContextMenu** remembers the **GoView** in which it is operating. Then the **MenuItem.Clicked** event handler can find the view by using the **GoContextMenu.FindView** method.

In GoDiagram Win when you specify the **Control.ContextMenu** property, the view would automatically bring up this context menu when the user right (context) clicks anywhere in the window. **GoToolContext** disables this behavior when the right click is on an object, to avoid interfering with the **ObjectContextClicked** event. Hence the **GoView.ContextMenu** property just specifies the default context menu, when the user clicks in the background or on some object that does not supply a custom context menu.

For GoDiagram Web to get a background/default context menu you can set the **GoView.DataRenderer.DefaultContextMenu** property, or override **GoViewDataRenderer.GetDefaultContextMenu** if you want to be able to generate a new context menu upon each rendering.

Mouse Over Events

Similarly, in GoDiagram Win when the mouse moves without any mouse button being held down, the **GoToolManager** tool invokes the **GoView.DoMouseOver** method. This in turn calls **DoToolTipObject** on the object (perhaps nothing/null) at the mouse event point, then **GoObject.OnMouseOver** on the object and its parents until **OnMouseOver** returns true, and finally **DoDefaultCursor** if no **OnMouseOver** call handled the mouse over event. This behavior ensures that every object will have a chance to display a tool tip and to have custom behavior in the **OnMouseOver** method. **GoHandle** objects, for example, may change the cursor in the **OnMouseOver** method.

DoMouseOver is also responsible for calling **DetectHover**, which uses a **Timer** to see if some sort of hover event needs to be raised. Again, only GoDiagram Win supports mouse-over and hover, because those concepts are not supported by the Compact Framework or Web Forms.

DoToolTipObject is organized in the same manner as the other **Do…Click** methods—it calls **GetToolTip** on the object and its parents until **GetToolTip** returns a non-null string. This string is displayed by the view's **ToolTip** object. You can turn off all tooltips by simply setting the **GoView.ToolTip** property to nothing/null.

Note that the **GoNode.ToolTipText** property provides an implementation of tooltip strings for all instances of **GoNode**. You can just set this property when

the tooltip information is constant for each node, or you can override getting this property to compute the string each time. If you want to display tooltips for other objects, such as links or ports, you will need to override the **GoObject.GetToolTip** method to return a string.

GoLink, **GoLabeledLink**, and **GoView** also implement the **ToolTipText** property. For **GoView**, the **ToolTipText** property determines the default tooltip for the whole view.

GoToolManager, besides calling **GoView.DoMouseOver**, also calls **GoView.DoObjectEnterLeave** if the document object immediately under the mouse point changes. This allows **GoView.ObjectEnterLeave** event handlers to update UI considering the "current" object(s) where the mouse is, and allows overrides of **GoObject.OnEnterLeave** to perform similar actions.

Disabling Functionality

Views also implement the **IGoLayerAbilities** interface, which defines the properties and methods used by Go to determine if the user may perform certain operations. These are:

- **CanSelectObjects**, **AllowSelect**
- **CanMoveObjects**, **AllowMove**
- **CanCopyObjects**, **AllowCopy**
- **CanResizeObjects**, **AllowResize**
- **CanReshapeObjects**, **AllowReshape**
- **CanDeleteObjects**, **AllowDelete**
- **CanInsertObjects**, **AllowInsert**
- **CanLinkObjects**, **AllowLink**
- **CanEditObjects**, **AllowEdit**

Setting any of the **Allow…** properties to false will disable the default behavior that allows the user to do that operation. Of course if any of the corresponding **Can…** methods on the object, its layer, or its document return false, the behavior is also disabled.

For convenience, the **SetModifiable** method allows one to set the move, resize, reshape, delete, insert, link, and edit ability properties all at once. Because there is such fine granularity on limiting user behavior, there is no **Modifiable** property.

Drag-and-Drop, Moving and Copying

The **GoToolDragging** class implements dragging behavior. For the view to make the dragging tool the current tool, at least one of the following **GoView** methods or properties must be true during a mouse drag: **CanMoveObjects**, **CanCopyObjects**, or **AllowDragOut**.

Remember that dragging to or from other windows, and interactive behavior during dragging, is only possible with GoDiagram Win. With both the Compact Framework and Web Forms, these events cannot occur and many of the associated methods are not defined. However, all of the dragging behavior that occurs within a **GoView** is handled by **GoToolDragging** in a manner that is the same for both Windows Forms and Web Forms.

Within a view, a drag moves the selected objects; between views a drag and drop copies the selected objects, and from another window the view can decide to accept the drop and to handle it in an application specific manner. If the user cancels a drag within a **GoView**, the selected objects are restored to their original locations.

For internal drag-and-drops, those that start and end within the same **GoView**, the default behavior is to move the selected objects. The **DragsRealtime** property controls whether the actual selected objects are moved along with the mouse, or whether an image of the selected objects is moved, leaving the selection in place until the move is completed. This image is part of the **DragSelection** in **GoToolDragging**. The default value for **GoView.DragsRealtime** is false.

GoToolDragging calls **GoView.MoveSelection** to perform the moving of the selection. Each object move causes a **GoDocument.Changed** event indicating that an object's bounding rectangle has changed. When **GoView.DragsRealtime** is true, there will be a lot of **Changed** events even before the final moves associated with the completion of the move gesture due to a drop. Setting **GoView.DragsRealtime** to false is more efficient when an undo manager is in effect, because all of the intermediate positions are not saved. After the move is complete, the tool raises a **GoView.SelectionMoved** event.

When the user holds down the CTRL key during a drag, the view prepares to copy the selection rather than move it. Because the selection is not copied until the user completes the drag, the **DragSelection** with the image of the selected objects is always shown moving with the mouse.

GoToolDragging calls **GoView.CopySelection** to perform the copying of the selection. The copied objects are added to the view's document using **GoDocument.CopyFromCollection**. Each copy causes a **GoDocument.Changed** event indicating an object insertion. After the copy is complete, the tool raises a **GoView.SelectionCopied** event.

GoToolDragging also raises **GoView.ObjectEnterLeave** events and makes calls to **GoObject.OnEnterLeave** by calling **GoView.DoObjectEnterLeave**. That method is called frequently in Windows Forms (except not in GoDiagram Pocket!) as the user drags the mouse around. For Web Forms, such events only occur at the beginning and the end of the drag, since only the start and the end points of a drag are passed in by the JavaScript code running on the browser. Your **GoView.ObjectEnterLeave** event handler can detect whether a drag or a mouse-over is taking place by checking whether the **GoView.Tool** is a **GoToolDragging** or not.

GoToolDragging also raises **GoView.BackgroundSelectionDropReject** and **GoView.ObjectSelectionDropReject** events in GoDiagram Win by calling **GoView.DoSelectionDropReject** from **GoToolDragging.DoMouseMove**. If the event is cancelled by setting the **InputState** to **Cancel** (or if the **GoObject.OnSelectionDropReject** method returns true), the drop is disallowed. This supports interactive control over whether a drop is allowed at a particular location in a view, or over a particular document **GoObject**.

GoView.DoSelectionDropReject is also called by **GoToolDragging.DoMouseUp**, on all platforms. If it returns true, the drag operation is cancelled; if it returns false, the drag is finished and the drag calls the **GoView.DoSelectionDropped** method. This supports easier customization of the action to be performed on a drop, particularly when the drop occurs on on object.

It is a moderately common case to override **GoObject.OnSelectionDropped** for a class representing a "container", to add the dropped objects to the container. The dropped objects are accessible as the **GoView.Selection**; they will normally have been added as top-level objects to the layer(s) of the document. Your code can then decide how to add them to your "container" object.

And you can easily control whether the "container" accepts particular drops by overriding **GoObject.OnSelectionDropReject**, returning true when not allowed. You can not only examine this particular object upon which the drop might occur, but also the objects in the **GoView.Selection**, to decide if a drop might be acceptable.

External drag and drop in GoDiagram Win

In GoDiagram Win **SelectionDropReject** and **SelectionDropped** events occur not only for internal drag-and-drops using **GoToolDragging**, but also upon an external drop, in **GoView.DoExternalDrop**. They do not happen interactively during the external drag, because the **GoView.Selection** does not yet hold the objects that are going to be dropped—in fact those **GoObject**s will not yet exist. However, if you set **GoView.ExternalDragDropsOnEnter** and **GoView.DragsRealtime** to true, a drag enter event will call **DoExternalDrop**, thereby creating **GoObject**s and adding them to the document and populating the **GoView.Selection**. Further drags will then use the view's **GoToolDragging** tool

to actually continue and perhaps finish the drag-and-drop, thereby enabling interactive **SelectionDropReject** behavior as well as the proper positioning of objects according to any grids.

In GoDiagram Win you need to set the **GoView.AllowDrop** property to true to enable drag-and-drop behavior. (This is not specific to **GoView--**you need to do this for any Windows Forms **Control**.) This property enables the user to drop onto the view, whether the drag started in that view or in any other window.

GoView in GoDiagram Win adds the **AllowDragOut** property, which enables the user to drag something from the view out to a different window. The default value for this property is true, except in the **GoOverview** class, which also sets **AllowDrop** false.

To customize a view as a drop target from other controls, you'll want to override **DoExternalDrag** and **DoExternalDrop**. By default **DoExternalDrag** sets the **DragEventArgs Effect** based on whether **CanInsertObjects** returns true. By default **DoExternalDrop** handles an event data object format of **GoSelection**. The selection is copied into the document using **GoDocument.CopyFromCollection**, passing an offset so that the copied objects are near the drop point. You'll want to override **DoExternalDrop** if you need to to handle a different data format used by a source window.

The ProtoApp and Demo1 samples provide examples of how to drag-and-drop **TreeNodes** from a **TreeView** into a **GoView** that overrides the **DoExternalDrop** method.

If you need somewhat more extensive customization, you can just override all the standard **OnDragOver**, **OnQueryContinueDrag**, and **OnDragDrop** event handling methods.

Normally an external drag-and-drop will not raise **GoView.ObjectEnterLeave** events. However, if the drop would create **GoObject**s that could be dragged around, you can set **GoView.ExternalDragDropsOnEnter** and **GoView.DragsRealtime** to true. This will cause an external drag enter to actually perform the **DoExternalDrop** immediately; the resulting selection is then dragged around by the view's **GoToolDragging** tool.

Resizing

Views also have default behavior for resizing objects, as implemented by the **GoToolResizing** class. When the user does a mouse down on a resize selection handle of an object whose **CanResize** method returns true, the view makes the **GoToolResizing** tool current, thereby going into resizing mode. This causes the **GoToolResizing.DoResizing** method to be called while the mouse is dragging the selection handle. This method in turn calls the **GoObject.DoResize** method on the selected object. The object can then decide how to interpret the resize request.

GoObject's default behavior in GoDiagram Win is to draw an XOR box during the resizing, and to set the object's bounds when the resizing is done. (On the Compact Framework, where XOR drawing is not supported, GoDiagram Pocket will draw a gray rectangle.) If the object's **ResizesRealtime** property is true, the object's bounds are set continuously as the mouse moves.

A resize may change the aspect ratio of an object unless the **CanReshape** method returns false. Most objects have the **AllowReshape** property set to true, but **GoImage**s have this property false by default.

When the user holds down the SHIFT key during a resize, the resizing maintains the aspect ratio of the object, even if **CanReshape** returns true.

Note that it is the **GoObject.SelectionObject**'s **CanResize** method that must return true for an object to be resizable. The **SelectionObject** may be different from the selected object itself, particularly for groups that redirect selection handles to an object in the group.

After the resize is complete, the tool raises a **GoView.ObjectResized** event.

Linking

Another important **GoView** feature is support for the user creating **GoLink**s between ports by "dragging" from a **GoPort** to another one. The **GoToolLinkingNew** tool implements this feature. It becomes the view's current tool when the user does a mouse drag from a port for which one or both of the **IGoPort.CanLinkFrom** and **IGoPort.CanLinkTo** methods return true.

The **GoToolLinking.CanStart** predicate uses **IsValidFromPort** and **IsValidToPort** to see if the port under the mouse point will permit the user to start a new link. If so, the view creates two temporary ports located at that port and a temporary link between the temporary ports. While the user remains in this creating-a-new-link mode, one temporary port continuously moves to follow the mouse. Because the other temporary port remains at the original port and because the link is redrawn as the port follows the mouse, the user sees the temporary link connecting the original port with where the mouse is.

Furthermore the view checks the ports to which it could make a valid new link, by calling **IsValidLink** for all potential pairs of ports involving the original one. The default implementation of **IsValidLink** just asks the "from" port if it can be linked to the "to" port by calling **IGoPort.IsValidLink**; this allows the behavior to be overridden either in the port class or in the view's tool.

To make drawing links easier for the user, there is also the notion of "port gravity", a distance. The temporary port automatically snaps to the location of the closest valid port within the port gravity distance. The **GoView.PortGravity** property has a default value of 100.

Finally, when the user releases the mouse to create the link, the **DoNewLink** method is called. This method is responsible for creating the real **IGoLink** (that

may be a **GoLink** or a **GoLabeledLink**, by copying the value of the
GoView.NewLinkPrototype property) in the document's links layer connecting
the two ports. The temporary ports and link are discarded. **DoNewLink** also
raises the **GoView.LinkCreated** event.

If for some reason the link is not made, because the attempted link was invalid or
because the user cancelled the link drawing process, the **DoNoNewLink** method
is called. This allows views to clean up any other state or inform the user or do
some other default failure action.

Rubber Banding

When the user drags the mouse without starting on an object, i.e. in the
background, the **GoToolRubberBanding** tool is used instead of
GoToolDragging. The normal behavior is to select objects with a rectangle, but
you can easily override the behavior to do something else.
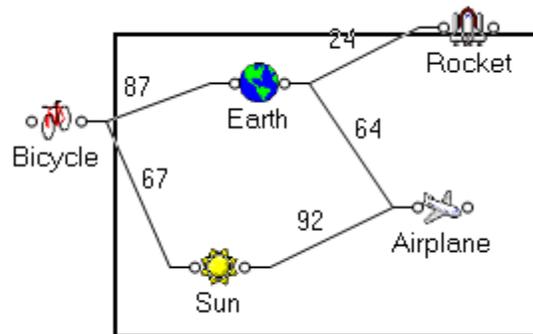
This simple tool just draws an XOR rectangle extending from the mouse down
point to the current mouse point. This drawing of course only happens on
Windows Forms; such interactive behavior does not occur with Web Forms.
When the user releases the mouse, the **DoRubberBand** method selects all
selectable top-level objects within the rectangle. The selection is performed by
GoView.SelectInRectangle.

The **GoToolZooming** tool is very similar to **GoToolRubberBanding**, but
instead changes the document position and scale of a view to correspond to the
rectangular box that was drawn. This tool is not normally used by **GoView**, but
it is used by **GoOverview**.

Clipboard

GoView supports copying the selection to and from the system clipboard; use the
GoView.EditCopy, **GoView.EditCut**, and **GoView.EditPaste** methods. These
methods depend on the document's **CopyFromCollection** method and use
GoDocument.DataFormat as the data format.

EditCopy and **EditCut** use the **GoView.CopyToClipboard** method to make a new instance of the view's document and copy in the collection of objects.

Similarly, **EditPaste** uses the **GoView.PasteFromClipboard** method to get the **GoDocument.DataFormat** data object from the system clipboard and copy its objects into the view's document.

You may wish to override the **GoView.CopyToClipboard** and **GoView.PasteFromClipboard** methods to handle additional data formats or to avoid using the **GoDocument** format.

The **GoView.CanEditCopy**, **GoView.CanEditCut**, and **GoView.CanEditPaste** predicates can be used to determine if their corresponding **Edit…** methods can be called, and thus to enable/disable parts of the user interface.

In-place Editing

Another handy feature that **GoView** offers is in-place text editing. Note that this interactive feature only applies to Windows Forms.

If a **GoText** object is editable, then clicking on it may put it into editing mode, where the user can change the string. This is accomplished by creating a temporary **GoControl** object in this view and having it be responsible for actually creating and displaying a **TextBox** and handling its editing completion or cancellation. The **GoControl** object is held as the **EditControl** property of the view.
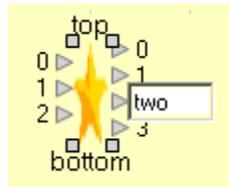
The **GoView.EditObject** method starts to edit any given object by calling **GoObject.DoBeginEdit**, assuming the view's **CanEditObjects** method returns true and the object's **CanEdit** method returns true. For the **GoText** class, **GoText.DoBeginEdit** does what is described in the previous paragraph.

Use **GoView.DoEndEdit** to stop any in-place editing in progress—this just calls **GoControl.DoEndEdit** on the view's **EditControl**, and then sets **GoView.EditControl** to nothing/null. In the **GoText** class, **DoEndEdit** raises a **GoView.ObjectEdited** event.

The **GoView.EditEdit** and **GoView.CanEditEdit** methods are similar to **GoView.EditCut**, **GoView.EditCopy**, **GoView.EditPaste**, and other **GoView Edit…** methods in providing easy-to-use methods for implementing and enabling user-interface commands. **GoView.EditEdit** just calls **GoView.EditObject** on the primary selection in order to get the work done.

Keyboard Commands

A view can accept keyboard focus and can respond to several keyboard commands by default. You can control whether there is any default key event handling by setting the **AllowKey** property, which defaults to true. You can also disable certain subsets of keys, by their functionality, by setting the **GoView.DisableKeys** property.

Normally, keyboard input is passed to the current tool by setting up a **GoInputEventArgs** and calling the tool's **DoKeyDown** method. All of the predefined tools interpret the ESCAPE key as a signal to stop the current tool. In this case, **DoKeyDown** just calls **DoCancelMouse** when the last input's key is ESCAPE, and **DoCancelMouse** just reset's the view's current **Tool** to the **DefaultTool**.

The normal default tool, **GoToolManager**, interprets additional keyboard commands as well.

Key	GoView.DisableKeys	Action
ESCAPE		**GoTool.DoCancelMouse**
DELETE	**Delete**	**GoView.EditDelete**
CTRL-A	**SelectAll**	**GoView.SelectAll**
CTRL-C	**Clipboard**	**GoView.EditCopy**
CTRL-X	**Clipboard**	**GoView.EditCut**
CTRL-V	**Clipboard**	**GoView.EditPaste**
F2	**Edit**	**GoView.EditEdit**
HOME	**Home**	**GoView.DocPosition** set to show the left edge of the document, top-left corner if CTRL
END	**End**	**GoView.DocPosition** set to show right edge of document, bottom-right corner if CTRL
PAGE-DOWN	**Page**	Scroll the view down by large increment, horizontally if SHIFT
PAGE-UP	**Page**	Scroll the view up by large increment, horizontally if SHIFT
CTRL-Z	**Undo**	**GoView.Undo**
CTRL-Y	**Undo**	**GoView.Redo**
letter or digit	**SelectsByFirstChar**	**GoView.SelectNextNode**
Arrow keys	**ArrowMove**	Move the selection in the given direction, one

		pixel at a time if CTRL
Arrow keys	**ArrowScroll**	Scroll the view in the given direction, one pixel at a time if CTRL

It is customary to use the F4 key to display a properties dialog or grid for the currently selected object (the primary selection). **GoView** and **GoToolManager** do not implement this because this functionality is much too application-specific. For GoDiagram Win, look at the ProtoApp sample for how you can use either a modal dialog or a properties grid for editing the properties of the current selection. For Web Forms, you will need to define your own custom property-editing forms.

VB.NET:
```vbnet
Protected Overrides Function IsInputKey(ByVal k As Keys) As Boolean
  If k = Keys.Down Or k = Keys.Up Or k = Keys.Left Or k = Keys.Right
  Then
    Return True
  End If
  Return MyBase.IsInputKey(k)
End Function
```

C#:
```csharp
protected override bool IsInputKey(Keys k) {
  if (k == Keys.Down || k == Keys.Up ||
      k == Keys.Left || k == Keys.Right)
    return true;
  return base.IsInputKey(k);
}
```

6. NODES

As noted previously, sets of Go primitive objects can be combined into higher-level grouped objects. One of the most common applications of this technique is in creating a "node" for a diagram, a node being a group with at least one port.

Ideally each application will want highly customized nodes. To get you started, Go provides a number of predefined node classes that have been shown to be useful in various kinds of applications. If they are not exactly what you are looking for, derive new classes from them to get exactly the appearance and behavior you seek.

Pictures of these objects are shown below with the descriptions.

- **GoBasicNode**, an elliptical or rectangular node with one port in the middle and an optional label

- **GoIconicNode**, the simplest node with an image and a text label and a single port

- **GoTextNode**, a node with four ports, one at each side and top and bottom, that displays some text with a background shape

- **GoMultiTextNode**, a node containing a list of objects (normally text) that has a port on each side of each list item and ports at the top and at the bottom

- **GoBoxNode**, a node containing an object, with a single port that is smart about connecting links to the closest side

- **GoSimpleNode**, a node with two ports, an icon, and a label

- **GoGeneralNode**, a node with any number of labeled ports on either side, an icon, and labels on the top and/or bottom

- **GoSubGraph**, a labeled node that contains a smaller diagram of individually selectable and movable nodes and links, that the user can collapse or expand in place

- **GoComment**, a group with no ports that displays some text

- **GoBalloon**, a balloon comment displaying text and pointing to an object

- **GoButton**, a group that looks and acts like a button, but is much lighter-weight than a real **Button** Control

- **GoListGroup**, a group that simply positions its children vertically or horizontally and provides a background, a border, and lines separating the children

One way to distinguish the different kinds of nodes is to consider how many ports they support and whether they display an image.

GoBasicNode, **GoIconicNode**, and **GoBoxNode** all are designed to have just one port. The actual points at which links connect to the port are dynamically computed.

GoTextNode is designed to have four ports. Even though in some cases a **GoTextNode** may look like a **GoBasicNode** or a **GoBoxNode**, links will always be connected at a particular port, no matter the direction the link comes from. Of course you can always remove one or more ports from a **GoTextNode**.

GoSimpleNode, **GoGeneralNode** and **GoMultiTextNode** have ports lined up on two sides; **GoSimpleNode** just has one on each side. You can vary the number of ports on a **GoGeneralNode** dynamically. Links are assumed to come into one side and go out the other. Both **GoSimpleNode** and **GoGeneralNode** support the **Orientation** property, which controls whether the ports are on the left and right, or on the top and bottom. The default is **Orientation.Horizontal**, so the ports are actually on the left and right.

GoMultiTextNode uses a **GoListGroup** to hold its main items. The number of ports in a **GoMultiTextNode** depends on the number of items; each item has a port on both sides, and there is one port at the top and one at the bottom.

GoIconicNode, **GoSimpleNode**, and **GoGeneralNode** all display an **Icon**, which is typically an instance of **GoImage**. The other nodes types do not, although some of them might contain images, such as **GoMultiTextNode** holding images instead of **GoText** objects.

GoComment, **GoBalloon**, **GoButton** and **GoListGroup** are not really nodes because they do not implement **IGoNode** nor do they contain ports.

GoBasicNode

A **GoBasicNode** has a shape (typically an ellipse or rectangle), a label, and a single port at the center of the shape. You can easily change the basic appearance of the node by setting its **Pen** and/or **Brush** properties, which just change those same properties on the **GoShape**. You can also replace the shape.
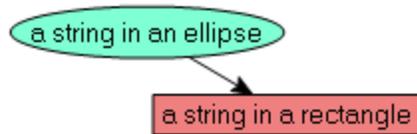
The label is only created when you set the **Text** property—its location relative to the shape is determined by the **LabelSpot** property, which defaults to **GoObject.MiddleTop**, placing the label centered above the ellipse.
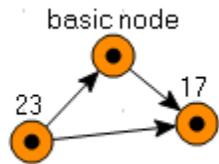
The natural location for a **GoBasicNode** is at the center of the shape, rather than at the top-left corner. Thus the location for a **GoBasicNode** is the same no matter where the label is.

If you replace the **Shape** object, a number of its properties are automatically copied from the original **Shape** object to the new one. These include the following properties of **GoObject**: **Center**, **Selectable**, **Resizable**, **Reshapable**, **ResizesRealtime**, and **Shadowed**. If you want to both replace the **GoBasicNode.Shape** and change any of these properties from the default, you will need to set the properties of the shape *after* setting the **GoBasicNode.Shape** property.

An easy way to replace the shape is to use a **GoDrawing** by specifying a **GoFigure** as an argument to the constructor:

```
GoBasicNode n = new GoBasicNode(GoFigure.ManualOperation);
n.Text = "some\nprocedure";
n.Label.Multiline = true;
doc.Add(n);
```
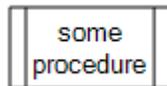
Please note that although using a **GoDrawing** is convenient for specifying different kinds of shapes, it is less efficient in space and rendering time than using a **GoEllipse** or a **GoRectangle**.

The link point for links at a **GoBasicNode** will be on the edge of the shape where the stroke of the link to the center of the shape intersects the shape.

Because there is no clearly indicated direction for links at the only port, you may want to use arrowheads on the links to indicate the direction of each link. One

way of achieving the effect that links pointing at **GoBasicNode**s have arrowheads, is to notice whenever a link gets added to the document:

```
doc.Changed += new GoChangedEventHandler(this.myDoc_Changed);

protected void myDoc_Changed(Object sender,
                             GoChangedEventArgs evt) {
  if (evt.Hint == GoLayer.InsertedObject &&
    evt.GoObject is GoLabeledLink) {
    GoLabeledLink l = (GoLabeledLink)evt.GoObject;
    if (l.ToPort != null &&
        l.ToPort.Node is GoBasicNode)
      l.RealLink.ToArrow = true;
  }
}
```

Instead of adding a document **Changed** event handler, an alternate (equivalent) way to get notification of events from a **GoDocument** is to create a subclass of **GoView** and override the **OnDocumentChanged** method, which is **GoView**'s event handler for document changes.

If you just want to check when the user draws a link, rather than when in all cases programmatic code creates a link and adds it to a document, you can instead add a **GoView.LinkCreated** event handler. Again, if you are inheriting from **GoView**, you could do the same thing by overriding **GoView.OnLinkCreated** if you wish to change the already created link, or by overriding **GoView.CreateLink** to control how the link is created:

```
public override virtual CreateLink(IGoPort from, IGoPort to) {
  GoLabeledLink l = new GoLabeledLink();
  l.FromPort = from;
  l.ToPort = to;
  if (to.Node is GoBasicNode)
    l.RealLink.ToArrow = true;
  this.Document.LinksLayer.Add(l);
  return l;
}
```

There is a special appearance for **GoBasicNode** when the **LabelSpot** is **Middle**. Then the label is indeed positioned at the center of the ellipse. But the ellipse is automatically resized to fit the text. You can control how much space there is around the text by setting the **MiddleLabelMargin** property.

If you want the node's **Shape** to remain at a fixed size, even when the text changes, you can set the **AutoResizes** property to false (the default is true).

The port, which would normally be visible at the center, becomes transparent and sized as large as the ellipse. Users can then start drawing a link from such a **GoBasicNode** with a mouse press and drag along the edge of the ellipse.
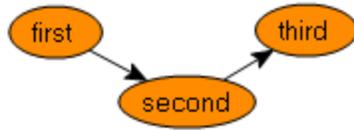
A **GoBasicNode** with the label in the middle is very similar looking to a **GoTextNode** with the same kind of shape as its background shape. However, the **GoBasicNode** only has one large port; the **GoTextNode** has up to four small ports—one on each side.

GoIconicNode

A **GoIconicNode** is the simplest node that has an icon. It has a text **Label** and a single small **Port**. The port is centered on the icon.

A **GoIconicNode** is convenient to use when there are simple relationships between the nodes and you only expect to create links programmatically. However, by default the user can draw new links starting at the **Port**.

The **Icon** can be any kind of **GoObject**, but is normally an instance of **GoImage**. After constructing a **GoIconicNode** you should call the **Initialize** method to specify the image from a **ResourceManager** or file. For Windows Forms, there is an overloaded **Initialize** method that takes an **ImageList**. Finally, if you call the **Initialize** method with both a null value for the **ResourceManager** and a null value for the **Name**, the **Initialize** method will allocate a **GoDrawing** instead of a **GoImage.** You can then set the **GoIconicNode.Figure** property as well as initialize any other properties of the **GoIconicNode.Shape**, such as its **Size** or **BrushColor**.

The above iconic nodes were constructed using the following code:

```
GoIconicNode n = new GoIconicNode();
n.Initialize(null, "star.gif", "star");
doc.Add(n);
```

That assumes that there is a **GoImage.DefaultResourceManager** defined that contains a "star.gif".

You can also make use of the various predefined **GoFigure**s:

```
GoIconicNode n = new GoIconicNode();
n.Initialize(null, null, "drawing");
n.Figure = GoFigure.FireHazard;
n.Shape.BrushColor = Color.Red;
n.Shape.BrushForeColor = Color.Orange;
```

```
n.Shape.BrushStyle = GoBrushStyle.SimpleGradientVertical;
n.Shape.Resizable = false;
doc.Add(n);
```

This produces the following result:

The label is normally positioned below the icon. If you turn on the **DraggableLabel** property, users will be able to move the label freely relative to the icon. This is handy when the user wants to avoid visual conflicts between the label and the node's links. You can also set the **LabelOffset** property programmatically.

The **MultiPortNode** example class in the Demo1 example inherits from **GoIconicNode** to provide an arbitrary number of ports at arbitrary positions on the icon.

GoTextNode

A **GoTextNode** is a relatively simple node class that displays text inside a rectangle with a port on each side of the rectangle. It is structurally similar to a **GoComment** except that it also has four ports, **TopPort**, **RightPort**, **BottomPort**, and **LeftPort** that are positioned at the middle of the edges of the node.
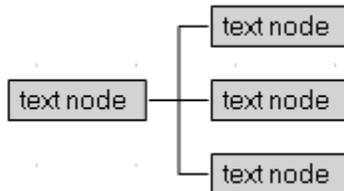
When the text string is changed, it automatically resizes the rectangle and moves the ports appropriately. The text supports multiple lines. By default it is not **Editable**, but by changing that property on the text label the Windows Forms user can single-click on a **TextNode** and start editing the text string. If the whole node is **Editable** and selected, then by default the user's F2 key will start editing the text label.

If you want the node's **Background** object to remain at a fixed size, even when the text changes, you can set the **AutoResizes** property to false (the default is true).

Because **GoNode** implements the **IGoLabeledNode** interface, the node's **Label** property is overridden to return the **GoText** used to display the text, and thus the node's **Text** property is the label's text string.

By default the background is an instance of **GoRectangle** and is the value of the **Background** property. However, you may use other kinds of **GoObject**s as the background for the text—using **GoRoundedRectangle** is common. You can either set the **Background** property explicitly, or you can override the **CreateBackground** method. If you override **CreateBackground**, you might do something like:

```
public override GoObject CreateBackground() {
  GoRoundedRectangle r = new GoRoundedRectangle();
  r.Selectable = false;
  r.PenColor = Color.Blue;
  r.BrushColor = Color.LightBlue;
  r.Shadowed = true;
  return r;
}
```

Depending on the shape of the background object, you may need to adjust the **TopLeftMargin** and **BottomRightMargin** properties to leave enough space for the text.

Another way to replace the **Background** shape is to use the **GoTextNode** constructor that takes a **GoFigure** as an argument. The **Background** is allocated as a new **GoDrawing** showing that **Figure**. For example:

```
GoTextNode n = new GoTextNode(GoFigure.Database);
n.Text = "\nsome database operation";
n.Label.Alignment = GoObject.Middle;
n.Label.Wrapping = true;
n.Label.WrappingWidth = 100;
doc.Add(n);
```

Produces the following kind of node:
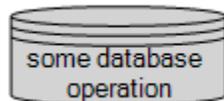
Unlike many other instances of ports, the ports on a **GoTextNode** are of **GoPortStyle.None**, so they have no appearance yet behave normally. You can remove ones you don't need by setting the corresponding property to nothing/null. You can disable them individually from letting users draw links to or from them by setting their **IsValidFrom** and/or **IsValidTo** properties to false. Or you can disable all linking by setting the **AllowsLink** property to false on the document or the view.

GoMultiTextNode

A **GoMultiTextNode** is useful when you wish to display a number of text items in a list, each with associated ports.
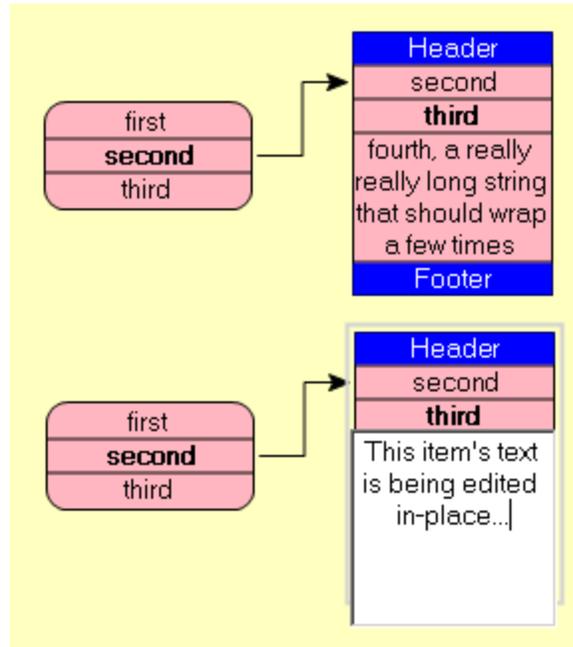
It is easy to add strings to the list—just call the **AddString** method. By default this will create a **GoText** that is not **Selectable**, is middle-aligned, supports multiple lines and text wrapping, and has the **DragsNode** property set to true.

This class uses an instance of the **GoListGroup** class to layout the items and draw a background, separator lines, and a border. But it also maintains an array of ports for the left side, an array of ports for the right side, the top port, and the bottom port. If you are navigating a graph and come to a port that is part of a **GoMultiTextNode**, you can find the port's associated item by calling **GoMultiTextNode.FindPortIndex** to return the zero-based item index.

Furthermore a **GoMultiTextNode** has its own notion of a how wide each item should be. The **ItemWidth** property, when positive, determines not only the width of newly created text items but their **WrappingWidth** too.

The **RecordNode** and **ObjectNode** classes are examples of how **GoMultiTextNode** can be customized for particular applications.

GoBoxNode

A **GoBoxNode** is also a relatively simple node, like **GoBasicNode**. It has a single port that is slightly larger than but centered behind the object (the **Body** property) that it displays. The body defaults to an instance of **GoText**, but you

can easily set it to be any **GoObject**. The Demo1 sample puts three text objects in the box by using a **GoListGroup** to hold the **GoText** objects and then setting the **Body** to be that group. If you intend to create many box nodes, you may find it wise to override the **CreateBody** method that is called by the constructor, to initialize and return the kind of object you would like to put in the box.

But the **GoBoxNodePort** is different from most ports because the link connection point and direction are determined dynamically according to the position of the port at the other end of the link. The connection point is always on the side closest to the link's other node, and it is always directed outward perpendicular to the port's side.
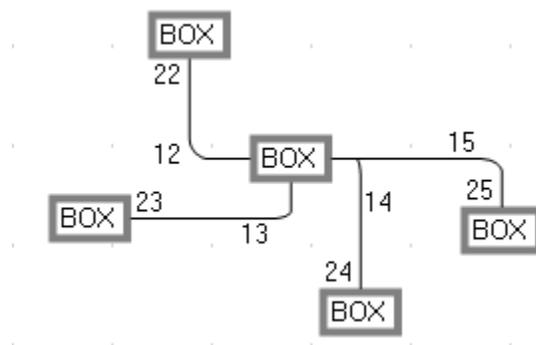
By default the link point is always in the middle of the closest side. However, by setting the **GoBoxNode.LinkPointsSpread** property to true, the links points will be spread evenly along each side.
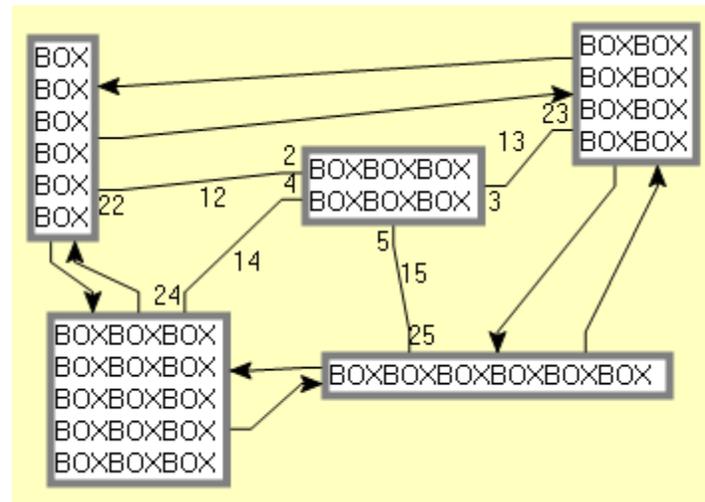
When the links are **Orthogonal**, the link points may be different:
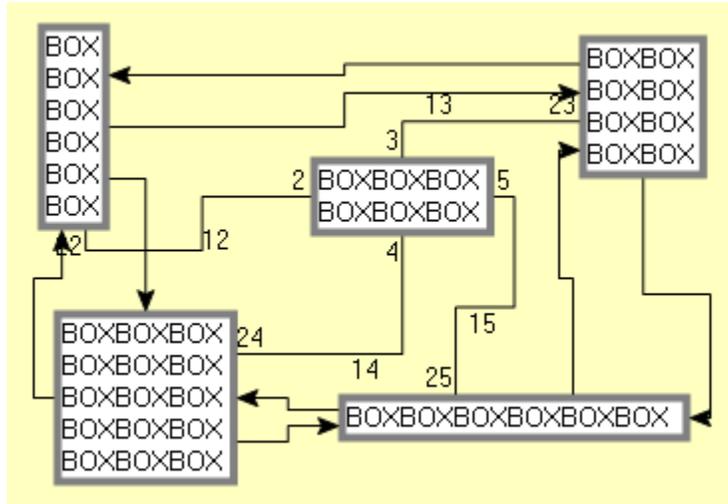
The **InfoNode** example classes in the Demo1 sample demonstrate one way to create nodes having various objects within a **GoGroup** that is the **GoBoxNode**'s **Body**.

GoSimpleNode

A **GoSimpleNode** has an icon, an editable label, and two ports for "input" and "output". It differs from a **GoTextNode** because the icon is the central object; for **GoTextNode**, the text is the central object.
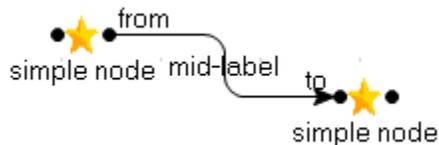
The resize behavior for **GoSimpleNode**s is that only the icon is resized. The **Icon**, in fact, is the node's **SelectionObject**. Furthermore, any resizing of the icon keeps the original aspect ratio of the icon, so that the icon does not appear distorted due to being stretched out sideways or up-and-down. The icon is normally a **GoImage**, in fact an instance of **GoNodeIcon**, which cooperates with this node to provide minimum and maximum icon size limits during resizing.

You can create a **GoSimpleNode** by allocating it and then calling **Initialize**. The choice of overloaded **Initialize** methods determines if the icon is an image taken from an **ImageList**, from a **ResourceManager**, or from a disk file. If you pass both a null value for the **ResourceManager** argument and a null value for the **Name** argument, the **Initialize** method will allocate a **GoDrawing** instead of a **GoImage.** You can then set the properties of this **GoDrawing** by setting the **Figure** property and by setting other **GoShape** properties.

The above **GoSimpleNode**s were created by:

```
GoSimpleNode sn = new GoSimpleNode();
sn.Initialize(null, "star.gif", "simple node");
doc.Add(sn);
```

That assumes that there is a **GoImage.DefaultResourceManager** defined that contains a "star.gif".

You can also make use of the various predefined **GoFigure**s:

```
GoSimpleNode sn = new GoSimpleNode();
sn.Initialize(null, null, "label");
sn.Figure = GoFigure.Pentagon;
sn.Icon.Size = new SizeF(40, 40);
sn.Icon.Resizable = false;
sn.Shape.FillShapeGradient(Color.Orange);
sn.InPort.Style = GoPortStyle.Diamond;
sn.InPort.Pen = null;
sn.InPort.BrushColor = Color.Green;
sn.OutPort.Style = GoPortStyle.Diamond;
sn.OutPort.Pen = null;
sn.OutPort.BrushColor = Color.Green;
doc.Add(sn);
```

This results in a node appearing as:
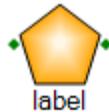
One way of changing the appearance of the node is to change its icon. If the image comes from an **ImageList** [Windows Forms only], you can set the index as follows:

```
aSimpleNode.Image.Index = 23
```

If the image comes from a **ResourceManager** or file:

```
aSimpleNode.Image.Name = "special.gif"
```

If the appearance comes from a **GoDrawing**, you can just set the **GoSimpleNode.Figure** property:

```
aSimpleNode.Figure = GoFigure.EightPointedBurst
```

Initializing a **GoSimpleNode** automatically creates the needed ports, and a label if the node name is not nothing (i.e., not a null reference).

You can easily change the appearance of a particular port by creating a single instance of the desired **GoObject** and then setting the **PortObject** (if the style is **GoPortStyle.Object**).

```
private static GoImage myStar = null;

public static GoImage GetStar() {
```

```
if (myStar == null) {
  myStar = new GoImage();
  myStar.Name = "star.gif";
}
return myStar;
}
```

```
. . .
  // assume aSimpleNode.InputPort.Style == GoPortStyle.Object
  if (. . .)  // want to change the port's appearance
    aSimpleNode.InputPort.PortObject = GetStar();
. . .
```

You can also change the size of each port individually:

```
aSimpleNode.InputPort.Size = new SizeF(10, 10)
```

By setting the **Orientation** property to **Orientation.Vertical**, the **InputPort** is positioned on top of the node, the **OutputPort** is positioned on the bottom**,** and the **Label** is positioned on the right side of the node.

The **GraphNode** example class used by the ProtoApp sample demonstrates keeping its label's text string unique within its document and adding a context menu command for editing the node's properties.

GoGeneralNode

A **GoGeneralNode** is similar to a **GoSimpleNode**, but with the following additional features:

- It supports a variable number of ports on each side.

- Each of those ports can have its own label, displaying the name of the port.

- There can be two labels for the whole node, on the top and on the bottom.

The **Image** or **Icon** properties are just like those of **GoSimpleNode**, regarding how they are initialized and how you can change their appearance.

Also like **GoSimpleNode**, you can set the **Orientation** property to **Orientation.Vertical** to switch the positions of the ports and the labels:

Note that the **LeftPort**s are now on the top of the node, the **RightPort**s are on the bottom of the node, the **TopLabel** is now on the left side, and the **BottomLabel** is now on the right side.

You can control whether the port labels are on the outside or on the inside of the ports by setting the **LeftPortLabelsInside** and **RightPortLabelsInside** properties. Here's a **GoGeneralNode** with the default settings, with the labels inside, and also with **Orientation** set to **Vertical**:
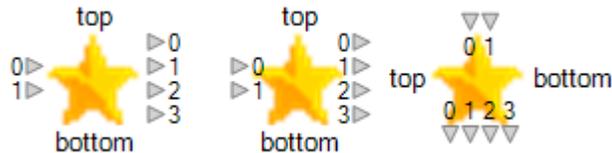
You can control how far the port labels are from their ports by setting the **LeftPortsLabelSpacing** and **RightPortsLabelSpacing** properties.

You can control how the ports are positioned relative to the edge of the **Icon** by setting the **LeftPortsAlignment** and **RightPortsAlignment** properties.

For example, the following code:

```
GoGeneralNode gn = new GoGeneralNode();
gn.Initialize(null, null, "top", "bottom", 4, 2);
gn.Figure = GoFigure.ManualOperation;
gn.Shape.Width = 50;
gn.Shape.FillSimpleGradient(Color.Orange);
doc.Add(gn);
```

combined with setting **Orientation** to either **Orientation.Horizontal** or **Orientation.Vertical**, and **LeftPortsAlignment** and **RightPortsAlignment** to **GoObject.TopLeft**, **Middle**, or **BottomRight**, results in nodes that appear as:
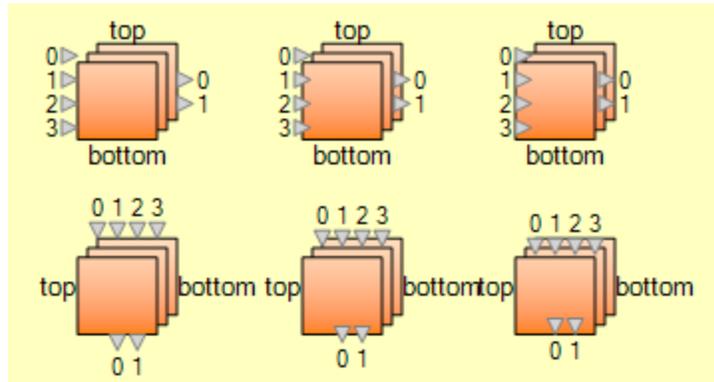
The **ColoredNode** example class, in the Demo1 subdirectory, uses a colored **GoRoundedRectangle** instead of a **GoImage** as the **Icon**, and uses ports that have a **GoPortStyle.Rectangle** style with different **Brush** colors.

The **LimitedNode** example class, in the Demo1 subdirectory, also demonstrates using context menu commands to add and remove ports from a node.

The **SequencedNode** example class, also in the Demo1 subdirectory, demonstrates how to extend a **GoGeneralNode** by adding a port at the top and a port at the bottom.

The **AutoLinkNode** example class, also in the Demo1 subdirectory, demonstrates a node with a special port in the middle of the icon. When the user links to that port, the appropriate left or right side port is added to the node and the link is completed to that new port. Furthermore, removing the last link from such side ports will automatically remove that port from the node.

GoSubGraph

Sometimes you have additional information associated with a node that is graphical in nature. One method of displaying such graphs is in a separate MDI child window. Typically, a double-click on the node should accomplish a drill-down by opening the detail window. The application would need to keep a hash table mapping nodes to MDI child windows, so that repeated double-clicks are able to restore and activate any existing window.

But there are times when you want to show the subgraph as part of the overall diagram. One way of doing this is to use the **GoSubGraph** class. This node allows its children to be individually selected and moved. Users can link nodes within a subgraph or between subgraph nodes and top-level nodes. As child nodes are moved, the subgraph does not move as a whole, but its **Bounds** are adjusted to include all of the children.
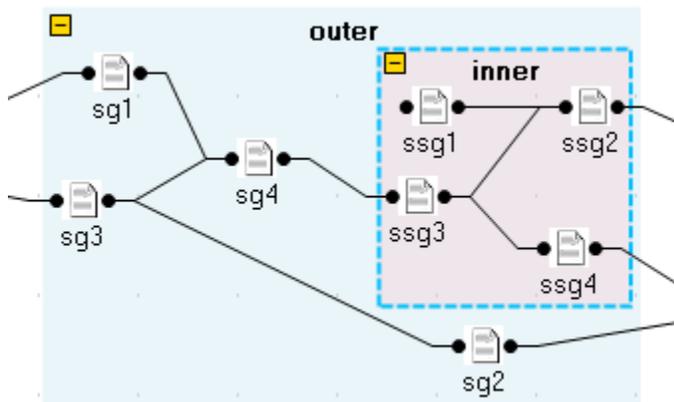
Each **GoSubGraph** has its own **BackgroundColor** and **Opacity** so that the boundaries of each subgraph are clear. There is also a **BorderPen** to provide an outline for the node's region, as shown by the "inner" subgraph above. The **Corner** property rounds off the four corners of the background. The **TopLeftMargin** and **BottomRightMargin** properties reserve extra space around the subgraph children. If you set the **GoObject.Resizable** property, the user will be able to adjust the margins interactively. The **GoObject.Shadowed** property also applies to the background.

The **PickableBackground** property controls whether a user's mouse press in the background of the subgraph will select the subgraph.

A **GoSubGraph** displays a text **Label** to help identify or describe the subgraph. You can specify the relative position of the label with the **LabelSpot** property. By default the **GoSubGraph** constructor calls **CreateLabel** to produce a **GoText** label that is placed at the **MiddleTop** spot.

A **GoSubGraph** has a handle that allows users to collapse or expand the subgraph with a click, just as they can with a node in a **TreeView**. The handle can also be used to move or copy the subgraph. A Control-click on a handle whose subgraph is collapsed will recursively expand any embedded subgraphs. Collapsing a subgraph makes all of the children invisible except the label. A collapsed subgraph will be large enough to hold the largest child along with the label.

Each **GoSubGraph** has a **Port** property that is normally used for connections to the node as a whole, rather than to any child node. However, by default there is no **Port**; you will need to set this property yourself, or override **CreatePort**, which is called during construction.

Collapsed subgraphs

When a subgraph is collapsed, it is typically a relatively small node. The subgraph children are made to be not **GoObject.Visible** and are recentered near

the **Handle**. The **Label** remains visible and is also centered in the collapsed node.

Its appearance can easily be customized using several **GoSubGraph** properties: **CollapsedTopLeftMargin**, **CollapsedBottomRightMargin**, **CollapsedCorner**, **CollapsedObject**, and **CollapsedLabelSpot**. The margin and corner properties are just like the non-"**Collapsed**" properties. **CollapsedObject** though is a replacement object that you can specify that is made **Visible** when the subgraph is collapsed. When there is a **CollapsedObject** (but by default this property is null/nothing), the **CollapsedLabelSpot** controls the position of the **Label** relative to the **CollapsedObject**. The **CollapsedObject** is made not **Visible** during an **Expand**.

Links in subgraphs

When the user moves or copies an object into the region of a **GoSubGraph**, it is *not* added to the subgraph group. You will need to implement your own policies and mechanisms for deciding when and how to add objects to subgraphs. The SubGraphApp sample provides an example implementation.

However, links are automatically added to subgraphs. When the user draws a link, **GoView.CreateLink** is called to construct a new link and add it to the document. The normal behavior is to see if both **FromPort** and **ToPort** belong to **GoSubGraph**s. If they do, the link is added to the **GoSubGraph** that contains both ports. If no such subgraph exists, the link is added to the **GoDocument.LinksLayer**, as a regular top-level object.

Links should be added to the first common parent of both end ports so that copying and autolayout of subgraphs works correctly. This is an exception to the convention that links should be added to the document's **LinksLayer**. The static **GoSubGraph.ReparentToCommonSubGraph** method will do this for you.

When you use the regular **GoNode** enumerators for iterating over the links (or ports or connected nodes) you will notice that it will include all of the links that are internal to the subgraph. You can simplify your programming by using the "**External**"-named enumerators, such as using **ExternalSources** to iterate over all of the **IGoNode**s that are source nodes to the subgraph.

Customizing Collapse and Expand

You can programmatically collapse or expand a subgraph by calling the **GoSubGraph.Collapse** or **GoSubGraph.Expand** methods. These methods are called when the user clicks on a **GoSubGraphHandle**. Both **Collapse** and **Expand** call a number of protected virtual methods that you can override in order to change the appearance and behavior of **GoSubGraph**.

Some examples are provided by the subclasses defined in the SubGraphApp sample. **CustomSubGraph** defines a fixed-size collapsed node. It also overrides **LayoutPort** so that the **Port** has the same **Bounds** as the whole

subgraph, rather than just being at the **Handle**. The **Port** itself uses a **GoBoxPort** to make connections "smarter" about where and how they connect at the port. **MultiPortSubGraph** customizes the margins to hold a variable number of ports (rather than having a single **Port**) and to allow the user to pick the **MultiPortSubGraph** by clicking or grabbing the thick margin. The appearance of each **MultiPortSubGraphPort** is implemented using shared **GoHexagon**s as the **GoPort.PortObject**s.

Further variations are provided commented-out in the example source code, such as keeping the **Handle** positioned at the far top-left corner of the subgraph (overlapping the margin), creating a **CollapsedObject** using a **GoImage**, and making the **Label** not **Visible** during a **Collapse**.

Collapse performs a number of steps that allow for customization. It may be easiest to present a (simplified) definition:

```
public virtual void Collapse() {
  if (this.State != GoSubGraphState.Expanded) return;
  if (!this.Collapsible) return;
  this.State = GoSubGraphState.Collapsing;
  this.Initializing = true;
  PrepareCollapse();
  // figure out how big the bounds will be, assuming any
  // nested subgraphs are collapsed, ignoring any collapsed margin
  SizeF maxsize = ComputeCollapsedSize(true);
  // ComputeCollapsedRectangle calls ComputeReferencePoint
  RectangleF cr = ComputeCollapsedRectangle(maxsize);
  foreach (GoObject obj in this) {
    SaveChildBounds(obj, cr);
  }
  foreach (GoObject obj in this) {
    CollapseChild(obj, cr);
  }
  FinishCollapse(cr);
  this.Initializing = false;
  this.State = GoSubGraphState.Collapsed;
  // make sure Handle, Port, Label are positioned correctly again
  LayoutChildren(null);
  this.InvalidBounds = true;
}
```

And here is a simplified definition of **Expand**:

```
public virtual void Expand() {
  if (this.State != GoSubGraphState.Collapsed) return;
  if (!this.Collapsible) return;
  this.State = GoSubGraphState.Expanding;
  this.Initializing = true;
```

```
PrepareExpand();
PointF hpos = ComputeReferencePoint();
// expand nodes (and other children), then links
foreach (GoObject obj in this) {
  if (!(obj is IGoLink)) {
    ExpandChild(obj, hpos);
  }
}
foreach (GoObject obj in this) {
  if (obj is IGoLink) {
    ExpandChild(obj, hpos);
  }
}
FinishExpand(hpos);
this.Initializing = false;
this.State = GoSubGraphState.Expanded;
// make sure Handle, Port, Label are positioned correctly again
LayoutChildren(null);
this.InvalidBounds = true;
}
```

GoCollapsibleHandle

You can easily implement your own classes that collapse and expand like a **GoSubGraph**, but that have other features. The **TreeAppNode** example class in the TreeApp sample demonstrates using a **GoCollapsibleHandle** for a completely different purpose than the **GoSubGraphHandle** of a **GoSubGraph**.
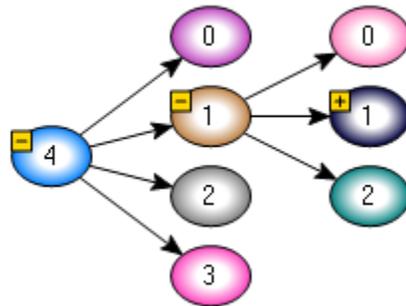
The technique is to add a **GoCollapsibleHandle** to your node or group that also implements the **IGoCollapsible** interface. The **GoCollapsibleHandle** object overrides **GoObject.OnSingleClick** to handle user's clicks to call its parent's **IGoCollapsible.Expand** and **IGoCollapsible.Collapse** methods.

Another example of using **GoCollapsibleHandle** is the **CollapsibleListGroup** example class. This group contains two **GoListGroup**s plus a **GoCollapsibleHandle** that controls the visibility of the two groups—one is **Visible** when the other one is not **Visible**.

GoCollapsibleHandle inherits from **GoRoundedRectangle**, so that you can set the **Pen** and the **Brush** and the **Corner** of the handle. But you can also specify the **Style** property to customize the internal appearance: values include: **GoCollapsibleHandleStyle.PlusMinus**, **TriangleRight**, **TriangleUp**, and **ChevronUp**.

GoSubGraphBase

GoSubGraph implements a particular model or style of grouping together a subset of a graph as a single node. It supports expanding and collapsing (including having a **Handle** and a **CollapsedObject**)**,** implements its own notions of a **Label** and a **Port**, and has its own conventions regarding margins and resizing.

However, you may want to implement your own "graph-container" nodes. Perhaps you don't like how **GoSubGraph** implements collapse and expand and overriding its methods is insufficient or too complicated for your application. You can do so by inheriting from the **GoSubGraphBase** class. This class (which inherits from **GoNode**) provides support for the additional graph traversal properties and methods that are specific to subgraphs. Inheriting from **GoSubGraphBase** will also make sure newly drawn and reconnected links are automatically reparented so that each link will be a child of the appropriate **GoSubGraphBase**.

GoComment

A **GoComment** is a very simple **GoGroup** that just has a text object with a background object. As the size of the text changes, the bounds of the comment adjust appropriately.
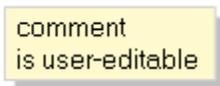
comment
is user-editable

The **Background** shape and **TopRightMargin** and **BottomRightMargin** properties are similar to those of a **GoTextNode**.

With Windows Forms there are two different predefined ways for users to start editing the text in-place. If **GoComment.Label.Editable** is true, a single-click will begin the in-place edit. If **GoComment.Editable** is true, an F2 key will begin the in-place edit of the selected comment. Of course, if you set both properties to false, the user will not be able to edit the text at all, unless you provide alternative mechanisms.

GoBalloon

A **GoBalloon** is a fancier **GoComment** that is associated with an object and points to that object.
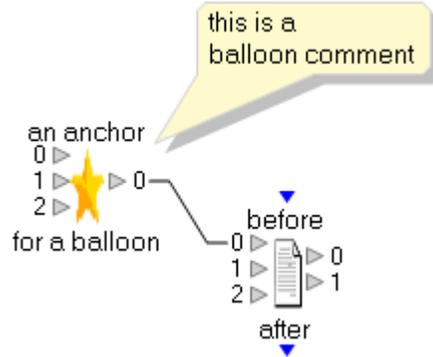
The **Anchor** property is the object that the balloon comment is about. Either the balloon comment or the anchor object can move independently.

The **BaseWidth** property controls how wide the base of the triangle is near the text label.

When the **Reanchorable** property is true and the balloon is selected, there is a special handle place at the point of the balloon, near the anchor object. Users can then change the balloon's **Anchor** property by dragging to another object. You can override the **PickNewAnchor** method to control what kinds of objects that are permitted to be new anchors, and whether the balloon can have no object as an anchor.

When the **Anchor** property is null/nothing, the point of the balloon is specified by the **UnanchoredOffset** property.

Your application code needs to decide what to do when the user deletes the object that is the anchor of a balloon. You might want to delete the comment, or you might keep the default behavior, which is to make the balloon unanchored.

GoButton

A **GoButton** has the appearance of a regular button. However, it is implemented as a **GoGroup**, containing a **GoText** label, a **GoImage** icon, and a **GoRectangle** background drawn with a simulated 3D border.

Unlike other **GoObject** classes, **GoButton** supports an event, the **Action** event, which is raised when the user does a mouse-down and a mouse-up within the button.

```
    aButton += new GoInputEventHandler(aButton_Pressed);
. . .

void aButton_Pressed(Object sender, GoInputEventArgs e) {
  String msg = "clicked on a GoButton";
  if (sender is GoButton) {
    msg += " labeled: ";
    msg += ((GoButton)sender).Text;
  }
  MessageBox.Show(msg);
}
```

GoButton implements the **IGoActionObject** interface to get mouse down, mouse move, and mouse up events from the **GoToolAction** tool without interfering with the standard mouse dragging and linking behaviors.

You can also disable a button by setting its **IGoActionObject.ActionEnabled** property to false.

Example Nodes

Other potentially useful node examples are provided in the sample directories, including:

- **ClassNode**, a **GoTextNode** that displays a type's name, properties and methods (in the Classier sample)
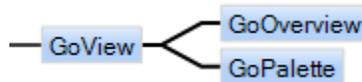
- **PersonNode**, a **GoTextNode** that displays a person's name (in the FamilyTree sample)
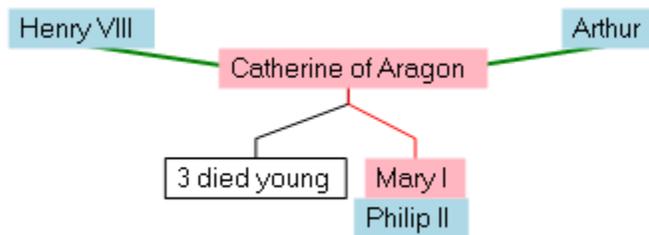
- **GraphNode**, a **GoSimpleNode** that has customized ports, context menu commands, and a unique label when added to a document (in the Demo1 sample)
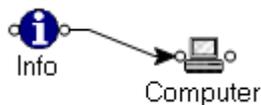
- **LitIconicNode**, a **GoIconicNode** that paints a highlight behind and around its Icon. In the Demo1 sample, the example class has implemented a **GoObject.OnEnterLeave** override to turn on the highlight when the mouse is over the node. In the following screen shot, the mouse is actually over "Lit 1".


- **ColoredNode**, a class in the Demo1 sample inheriting from **GoGeneralNode** that uses a colored **GoRoundedRectangle** instead of **GoImage** as the **Icon**, and whose ports are colored rectangles. Note that the default **Orientation** is **Orientation.Vertical**. Furthermore there is an additional label that is positioned in the middle of the icon. The standard two labels that **GoGeneralNode** provides are still available for use on either side of the node.

   The ports do not have labels, but they do have names. The user can see these names in tooltips when the user hovers the mouse over the ports. The colors of the ports are chosen at random in this example class, but you may want to modify the code to assign particular colors and/or port styles to help visually identify the individual ports.

- **LimitedNode**, a **GoGeneralNode** whose location is limited to a certain range in the X dimension, that has context menu commands for adding and removing ports, whose ports are limited to at most three links, and whose ports change color according to how many links are connected (in the Demo1 sample)
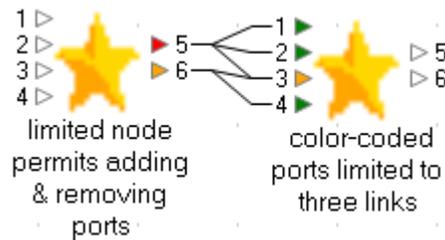

- **AutoLinkNode**, a **GoGeneralNode** with a special auto-linking port that automatically creates new ports as the user tries to link to it (in the

Demo1 sample)

auto-link

- **SequencedNode**, a simple extension of **GoGeneralNode** to give it extra ports at the top and at the bottom (in the Demo1 sample)
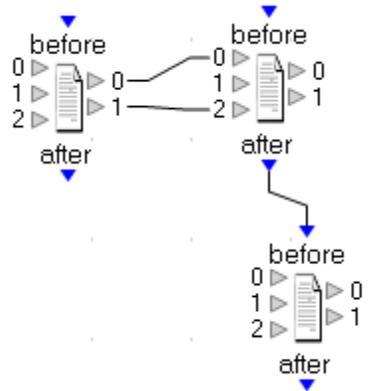
- **MultiPortNode**, a **GoIconicNode** extension that provides a variable number of ports at arbitrary positions on the icon. The **Label** is also repositionable by the user (in the Demo1 sample)
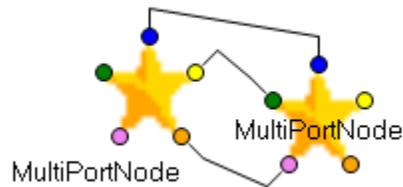

- **MultiTextNodeWithBack**, a **GoMultiTextNode** that has a separate **Background** object. For example, when the object is a **GoCylinder**:
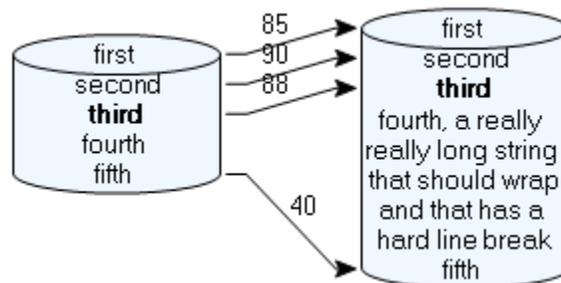
However, any **GoObject** can be used, such as a **GoImage**. This class is also yet another instructive example of how to add an object to a node class.

- **ScrollingMultiTextNode** is also a **GoMultiTextNode**, but demonstrates how to add a couple of **GoButton**s to control the **ListGroup**'s **GoListGroup.TopIndex** to get the items in the **GoListGroup** to scroll.
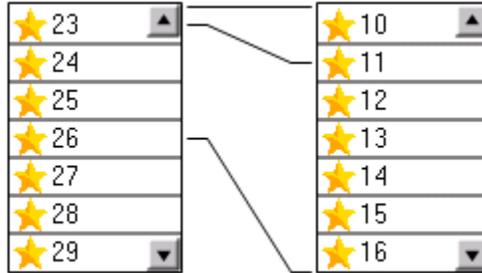
There are actually 100 items in each of the above example nodes, but the first 23 and last 70 are not **Visible** in the left node. Item 6 in the left node is connected to item 6 of the right node; 23 to 11, and 26 to 26. Each node can also be resized interactively, because the user actually resizes the **GoMultiTextNode.ListGroup**.

In Windows Forms if the user holds down the mouse button on a scroll button, the items will autoscroll after a delay. The user can increase the rate at which it autoscrolls by dragging the mouse vertically farther away from the button.

In Web Forms there is no autoscrolling, of course. However, the user can still increase how many items are scrolled by releasing the mouse far away vertically from the button.

- The **InfoNode** example class in the Demo1 sample demonstrate one way to create nodes having various objects within the **GoGroup** that is a **GoBoxNode**'s **Body**.

The parts include **GoText** objects to display text strings, some **GoShape**s to provide background or informative colored shapes, and even a **GoButton** that can handle user clicks. You can use **GoListGroup**s to organize the various parts of each node and provide separator lines.

Other **InfoNode** example classes demonstrate other ways of constructing more complex nodes.
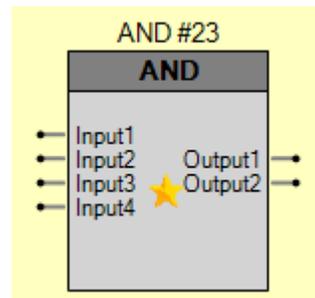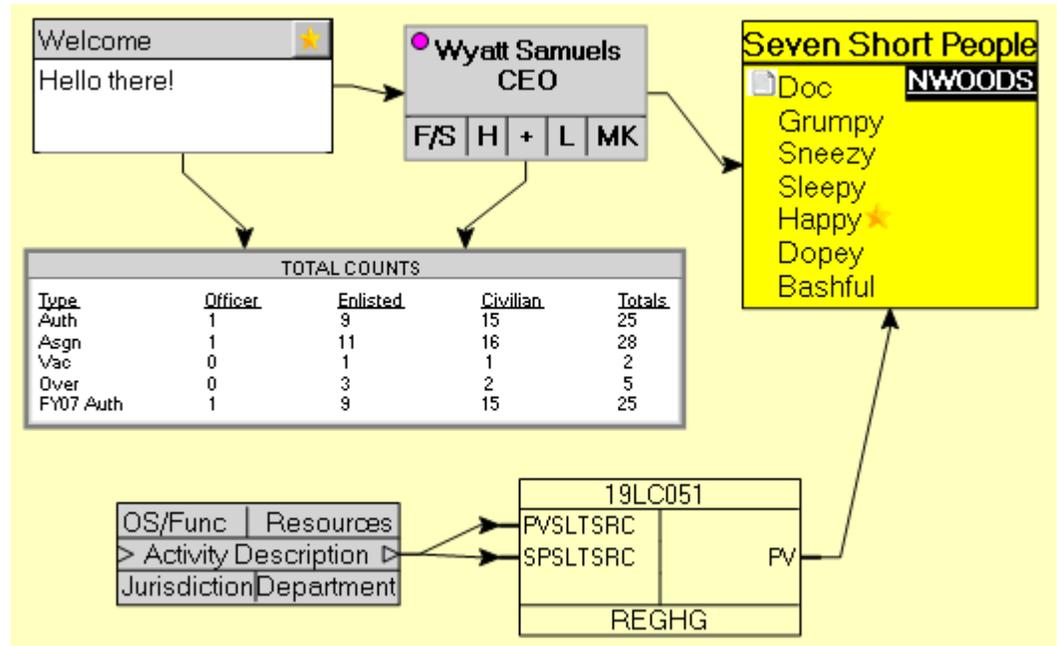
- **RecordNode**, a **GoMultiTextNode** whose items have a single port, a **SidewaysPort** that extends to both sides of each item. Although the items are typically just **GoText** objects, the **RecordItem** class is provided so that an item can have an image associated with it.
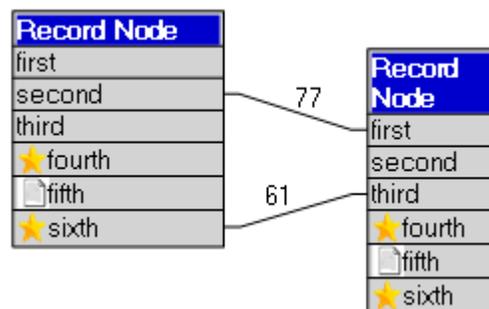
- **CollapsingRecordNode** is a more complicated and more sophisticated node that supports tree structure of its items by nesting **CollapsingRecordNodeItemList**s and by supporting indentation in each leaf **CollapsingRecordNodeItem**. (The implementation does not use **GoMultiTextNode** at all.)
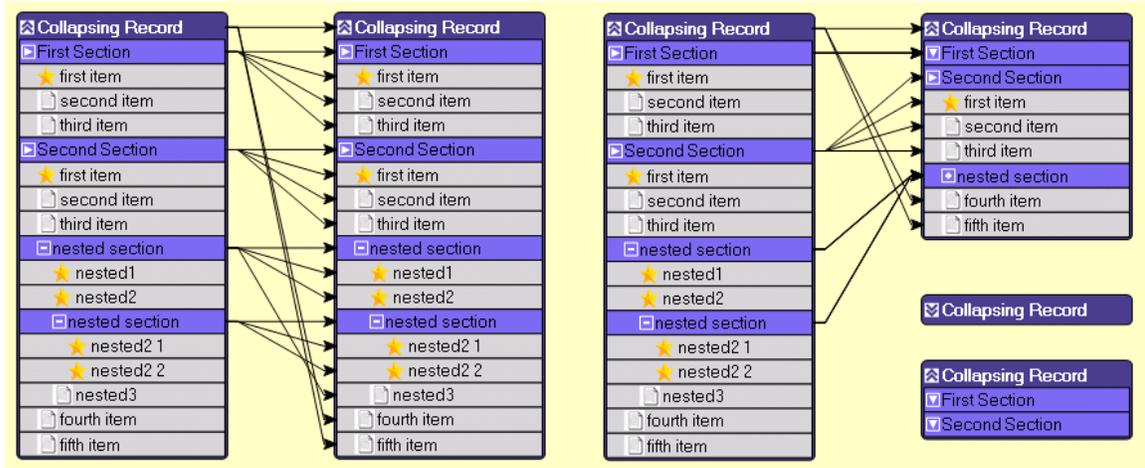
- **ClassDiagramNode** provides a node with a single port that displays partitioned collapsible lists of items. The whole node can also be collapsed, as shown with the chevron-style **GoCollapsibleHandle** that is located at the top-right corner of the node.
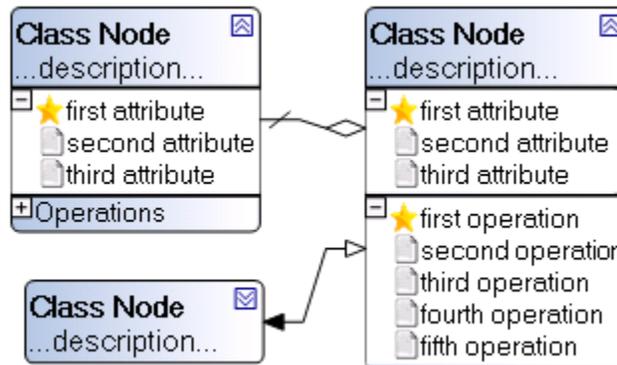
- **ObjectNode**, another **GoMultiTextNode** that uses slightly different **SidewaysPort**s, to represent an in-memory object. Some items are named references to other objects. The final item can be a **GoGroup** of **GoPort**s with links representing memory references to other objects.
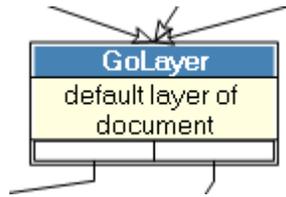
- **PinNode**, a node that has four prominent ports on each side of a rectangle with a text label, connected by lines. This class demonstrates a simple use of custom painting in the node—the **Paint** override draws two lines between the ports and the **GoRectangle** and **GoText** cover over the middle where the two lines intersect. The **GoText** wraps, but if it doesn't fit in the given space within the rectangle, the **StringTrimming** property specifies **StringTrimming.EllipsisCharacter** (but this feature is not available in .NET Compact Framework).
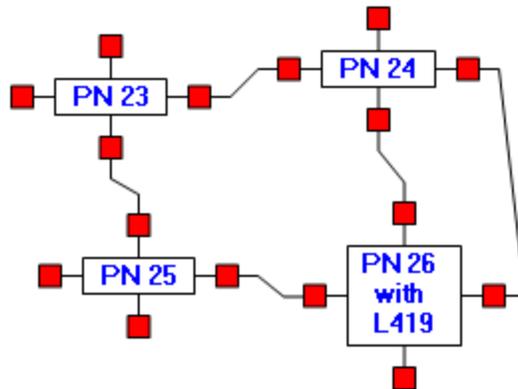
- **LinkLabel**, in the Processor sample, demonstrates how to customize a **GoText** class so that it can be dragged by the user and draw a line connecting the label with a spot on the link.
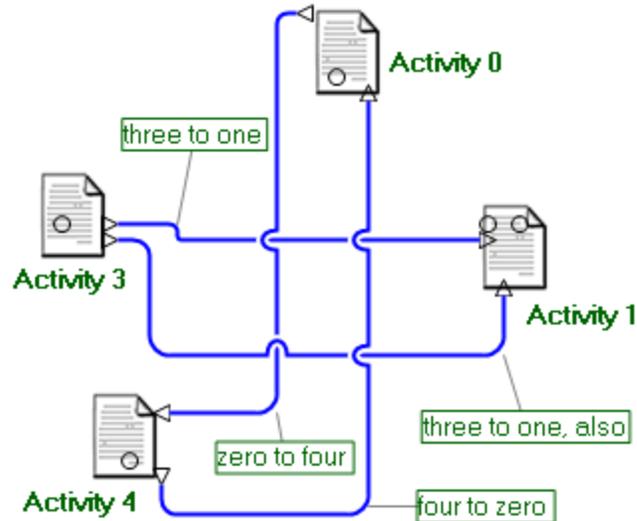
Also you may want to look at other classes such as:

- **GraphLink**, a labeled link with an arrowhead, whose middle label is initially a random number, and that has a context menu (in the Demo1 sample)
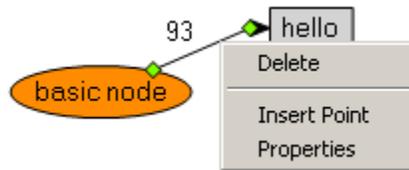
- **TubularRectangle**, a **GoRectangle** subclass that implements a custom **Paint** method. Note that unlike a **GoRoundedRectangle**, the sides are never quite straight, because 4 Bezier curves are used instead of 4 straight lines connected by 4 arcs.

- **RichText**, a **GoObject** class that displays formatted text (in the Demo1 sample)

- **AndShape**, **OrShape**, **HouseShape**, **OctagonalStar** as shapes in Demo1. You can specify the direction for the first three shapes.
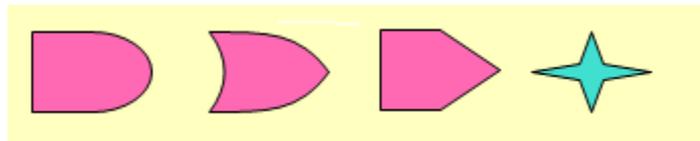
- **RectangleWithCheckBoxEditor**, in Demo1, demonstrates bringing up a **CheckBox** as an editor for an object, in this case a class inheriting from **GoRectangle**. The screenshot shows how it appears while the user is editing the object. Unchecking the checkbox and changing focus away from the checkbox would cause the object's brush color to change.
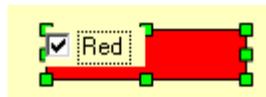
- **GradientColorLink**, in Demo1, makes use of a Pen that uses a linear gradient brush going from a color at one port to another color at the other port.

- **TwoColorLink**, also in Demo1, demonstrates customized drawing of the stroke of a link.
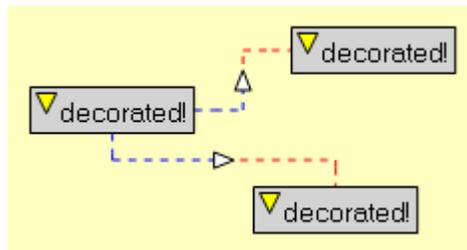
- **TriangleTextNode**, also in Demo1, demonstrates custom drawing in the **Paint** method of a node. (See above for screenshot.)

- Various **GoGroup**s, such as the **Rack** and **Shelf** and **Item** classes in Planogrammer, demonstrate that not everything need be a "node".
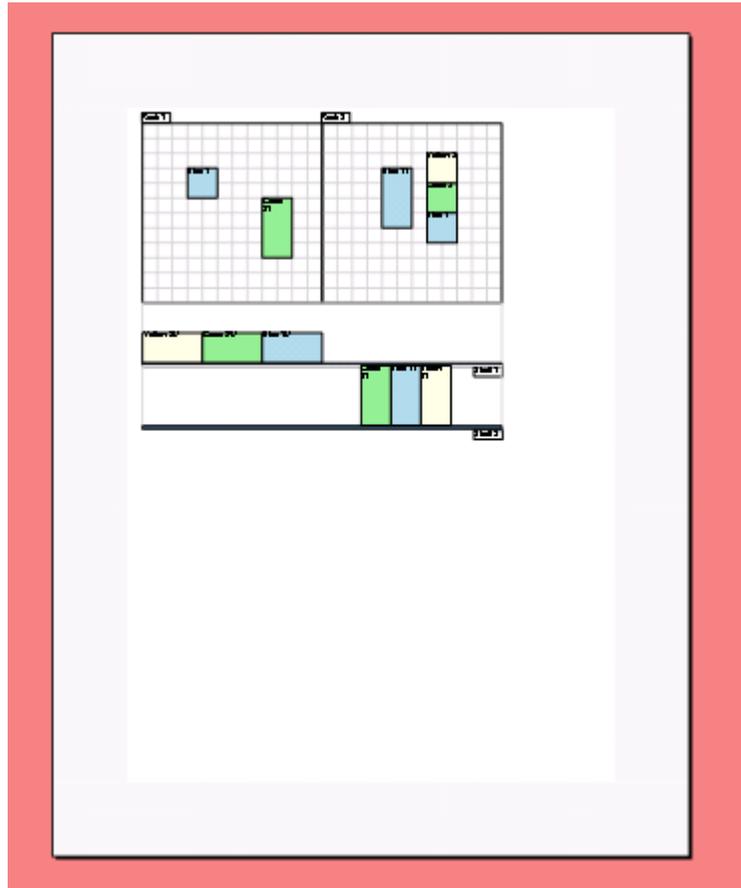
For all of these examples, be sure to look at the source code for more descriptions and details.

Example SubGraph Classes

TreeSubGraph

TreeSubGraph is a relatively simple example subclass of **GoSubGraph** that defines a minimal appearance for the node and defines a **Port** that is located at the **Handle**.
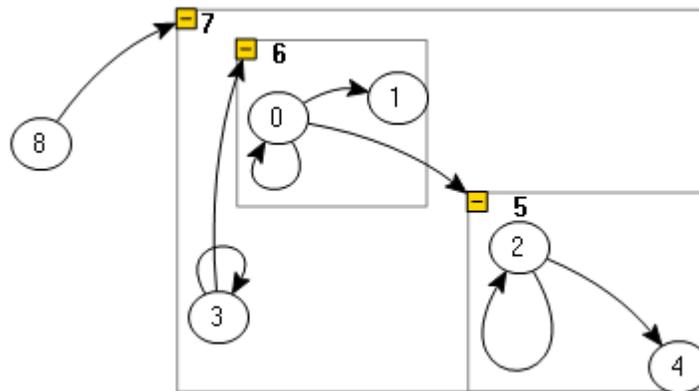
When the two inner nodes are collapsed, you see:
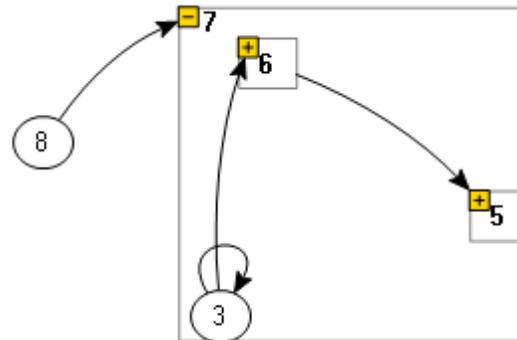
Collapsing the outer node results in:
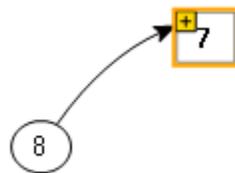
In this last screenshot, the collapsed node is selected.

CustomSubGraph

CustomSubGraph defines a **Port** that is implemented by a **GoBoxPort** with the same bounds as the whole subgraph. This ensures that links to or from the **Port** are always evenly spaced along the closer sides of the node.

The **Port** itself has a **Style** of **GoPortStyle.None**, so that it is not seen.

These example nodes also demonstrate moderately large values for the **Corner** and **Margin** properties, along with **Shadowed** being true.
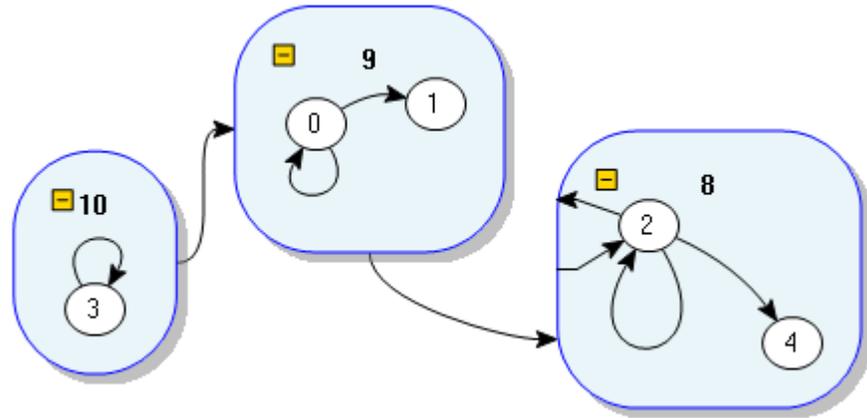
Links to the **Port** that come from within the subgraph are treated differently—the link direction is reversed, so that they connect directly from within the port, rather than having to go outside and turn around to point back inwards.

The same graph, with the nodes collapsed:
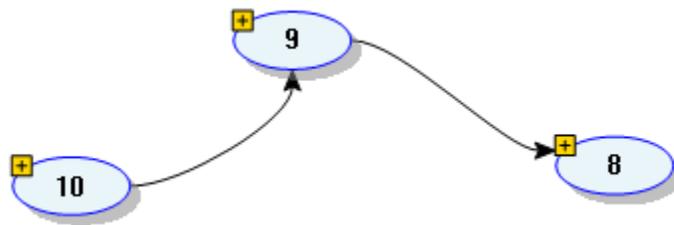
Although not shown in this screenshot because the labels are short strings, the **CustomSubGraph** has the ability to abbreviate the labels using ellipsis when they are collapsed so that collapsed nodes are always the same size.

MultiPortSubGraph

The **MultiPortSubGraph** class defines a collection of **MultiPortSubGraphPort**s that can be positioned along one of the sides of the whole node. In this example, these ports are rendered using **GoHexagon**s as the **PortObject**s, so that there are only four instances of **GoHexagon** that are shared by all **MultiPortSubGraphPort**s.
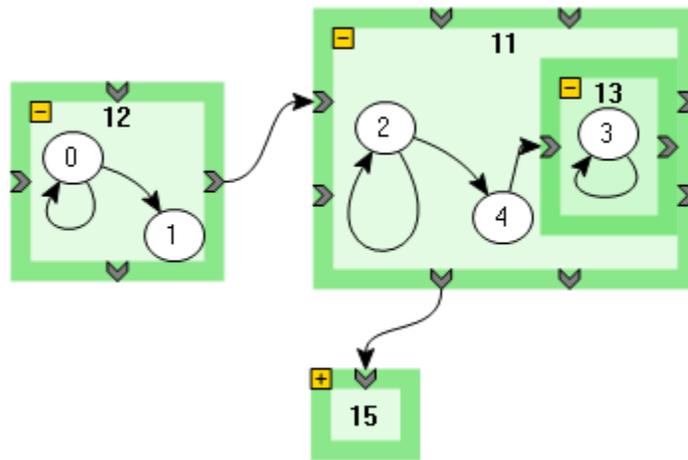
The same graph, with the nodes collapsed:
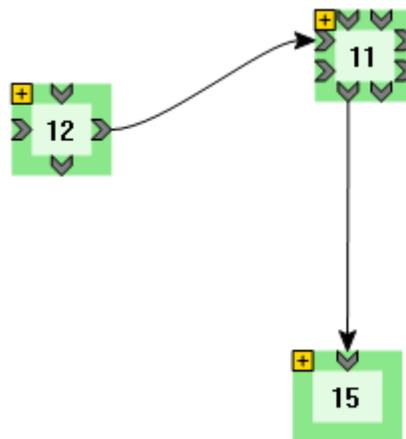
Note how the ports remain visible and are repositioned evenly on each side.

LimitingSubGraph

Normally there are no restrictions on the movement of child objects within a **GoSubGraph**. However, the **LimitingSubGraph** example class in Demo1 demonstrates how a **GoSubGraph** can implement **IGoDragSnapper** that affects how drags and resizes can be restricted to stay within the current subgraph border.

Furthermore each **LimitingSubGraph** contains two special markers that the user can drag around that are not limited by the snap-point behavior. This allows the user to resize the subgraph freely.

7. UNDO AND REDO

Go makes it easy for programmers to build graphical applications that display relationships between objects and that allow users to change those relationships with little effort. Because users can make massive changes so easily, a well-designed application should also allow users to reverse the consequences of unintended changes.

Go provides built-in support for undo and redo of all operations that modify the state of a **GoDocument**, including any **GoObject** contained in a document. This support comes primarily by a **GoUndoManager** that is associated with a document that implements support for undo and redo.

GoUndoManager provides several methods that your user-interface commands can call. Two methods actually perform changes: **Undo** and **Redo**. Two predicates determine whether these operations can be performed: **CanUndo** and **CanRedo**. For convenience, these methods are also implemented on **GoDocument**, delegating to the document's undo manager. And they are implemented on **GoView**, delegating to the view's document.

Go's built-in support for undo and redo only operates on the state that it knows about. If you subclass **GoDocument** or one of the **GoObject** classes and add any new properties or other state that you want to include in undo and redo operations, your code must follow certain conventions, as described below.

Implementing Undo and Redo support in your application

If you want to support undo and redo in your application, you will need to do five things:

- Raise a **GoDocument.Changed** event for any application-specific state change.

- Perform the undo and redo actions for any application-specific change by overriding the **ChangeValue** method.

- Set your document's **GoDocument.UndoManager** property.

- Declare groups of changes that the user will want to consider a single logical edit by wrapping state-modifying code with calls to **StartTransaction** and **FinishTransaction**.

- Implement the user-interface commands to allow users to perform an undo or a redo, with the appropriate appearance, if you want anything besides the support for Ctrl-Y and Ctrl-Z that **GoView** provides.

Go implements undo and redo support for all predefined document and object classes, including the node classes. If you do not extend the state of any documents, you do not need to do the first two steps above dealing with application-specific state.

The built-in support for undo in Go only applies to documents and document objects. Changes to views, such as selection and view position, are not considered to be edits to the document, and therefore are not tracked for undo and redo. Also, the **GoUndoManager** cannot track any changes to **GoObject**s that are not part of a document.

IGoUndoableEdit and GoChangedEventArgs

The basic concept for remembering state changes is the **IGoUndoableEdit**, an interface that describes an object that represents a change to a document and the ability to undo and redo that change.

A change to a document means that some part of the document's state has been altered. This includes changing the values of any properties of a document, adding **GoObject**s to a document, removing them, and changing any properties or parts of any document objects.

If you want to add undo and redo functionality to your application, you must make sure that your **GoDocument** and **GoObject** extensions faithfully signal any state changes by calling **GoDocument.RaiseChanged** or **GoObject.Changed** respectively, passing the old and new values. Your extensions must correspondingly implement the **CopyOldValueForUndo**, **CopyNewValueForRedo**, and **ChangeValue** methods as needed.

Not all document state need participate in this undo framework. However, you and your users must be willing to live with the inconsistencies that might result when the user makes a change and a later undo does not restore the state as they might expect. You may find that some state currently associated with a document really belongs in the application, in a view or in a form.

Extending GoDocument

The ProtoApp and Demo1 examples include a representative document extension: adding a **Location** property in the **GraphDoc** class. The class definition, with parts elided for clarity, looks like the following code:

VB.NET:

```vb
<Serializable()> Public Class GraphDoc
  Inherits GoDocument

  Public Sub New()
    MyBase.New()
    ' enable undo/redo memory for this document
    Me.UndoManager = New GoUndoManager()
  End Sub

  Public Property Location() As String
    Get
      Return myLocation
    End Get
    Set(ByVal Value As String)
      Dim old As String = myLocation
      If Not old = Value Then
        myLocation = Value
         ' don't raise the Changed event unless it really changed
        RaiseChanged(ChangedLocation, 0, Nothing,
                      0, old, NullRect,   ' pass the old value
                      0, Value, NullRect) ' pass the new value
      End If
    End Set
  End Property

  ' actually perform the undo or redo
  Public Overrides Sub ChangeValue(ByVal e As GoChangedEventArgs, ByVal
undo As Boolean)
    Select Case e.Hint
      Case ChangedLocation
        Me.Location = CType(e.GetValue(undo), String)
      Case Else
        MyBase.ChangeValue(e, undo)
    End Select
  End Sub

  ' Event hints
  Public Const ChangedLocation As Integer = LastHint + 23

  ' Document state
  Private myLocation As String = ""
End Class
```

C#:

```csharp
  [Serializable]
  public class GraphDoc : GoDocument {
    public GraphDocument() {
      // enable undo/redo memory for this document
      this.UndoManager = new GoUndoManager();
    }
```

```
    // Location property
    public String Location {
      get { return myLocation; }
      set {
        String old = this.Location;
        if (old != value) {
          myLocation = value;
          // don't raise the Changed event unless it really changed
          RaiseChanged(ChangedLocation, 0, null,
                       0, old, NullRect,    // pass the old value
                       0, value, NullRect); // pass the new value
        }
      }
    }

    // actually perform the undo or redo
    public override void ChangeValue(GoChangedEventArgs e,
                                     bool undo) {
      switch (e.Hint) {
        case ChangedLocation:
          this.Location = (String)e.GetValue(undo);
          return;
        default:
          base.ChangeValue(e, undo);
          return;
      }
    }

    // Event hints
    public const int ChangedLocation = LastHint+1;

    // Document state
    private String myLocation = "";
}
```

Note that setting the **Location** property makes sure that there really is a change before setting the internal **myLocation** field and then calling **RaiseChanged**.

The call to **RaiseChanged** passes a hint, **ChangedLocation**, the old property value (**old**) and the new property value (**value**). It is important that the hint be unique within the class and all of its superclasses.

It is also required that the call to **RaiseChanged** occur *after* the change has happened, and that the update event handler is able to retrieve the previous value. Normally the previous value is passed along as part of the document changed event, so this is not a problem. The reason for the requirement that the previous value be accessible is that the document changed handler responsible for undo and redo needs to record the values both before and after an edit. These values

are used to construct a **GoChangedEventArgs**, which implements **IGoUndoableEdit**.

A **GoChangedEventArgs** is constructed using the before and after values from the change's call to **RaiseChanged**. In most cases the old value is just fine to remember as is; however if the value is a reference to an object that might be modified by further edits, it is important that the **GoChangedEventArgs** keeps a true copy of the old value, rather than just a reference to something whose relevant state may be changed. Thus the **GoDocumentChangedEdit** constructor calls the **CopyOldValueForUndo** method, which allows the class to decide whether the previous value needs to be copied for safekeeping. Many classes do not have any kinds of changes where the previous value will need to be copied, so they do not bother to override **CopyOldValueForUndo**.

Similarly, **GoChangedEventArgs** gets the new value by calling the **CopyNewValueForRedo** method. Again, the new value passed in the call to **RaiseChanged** usually obeys the value semantics needed for the remembering for undo and redo. If this is not the case, each class that extends the undoable document state must override this method to handle the class-specific changes to remember the new (current) values. In the example above, the location is stored as an immutable string, so there is no need to override **CopyNewValueForRedo**.

Finally, each class must override **ChangeValue** in order to perform the undo or redo, depending on the value of the boolean argument. For convenience the **GoChangedEventArgs.GetValue** method also takes the same **undo** parameter to decide whether to return the old/before value or the new/after value. In the example above, the method just needs to set the **Location** property to effect the change. For change hints not belonging to this class, the method calls the base method.

For efficiency and for convenience the old and new values of a **GoChangedEventArgs** are not simply **Object**s, but a pairing of an integer, an **Object**, and a **RectangleF**. For those properties that can be represented efficiently by an integer, you can use that instead of boxing the integer by creating an **Integer**. For property value types that are **PointF**, **SizeF**, **RectangleF**, or even just a **single** or **float**, you can use the **RectangleF** argument to avoid boxing those values. For those properties that can be conveniently represented by an integer and an object, you can use more than one of the values. For example a change to an element of a vector can be represented using both the integer and **Object** parameters for both the old and the new values.

Extending GoObject subclasses

For a change to a **GoObject**, the **Hint** is **GoLayer.ChangedObject**. However, there is no way for the document to know how to remember the old or new values for any particular object, nor how to perform that particular state transition. Instead those responsibilities are transferred to **GoObject**, which has

the same **CopyOldValueForUndo**, **CopyNewValueForRedo**, and **ChangeValue** methods.

The implementation is very similar to that for adding properties to documents. What follows is the definition of **LimitedPort** class, stripped down to essentials regarding the **MaxLinks** property, which governs the maximum number of links allowed on the port.

```
public class LimitedPort : GoPort {
  public int MaxLinks {
    get { return myMaxLinks; }
    set {
      int old = this.MaxLinks;
      if (old != value && value >= 0) {
        myMaxLinks = value;
        Changed(ChangedMaxLinks,
                old, null, NullRect,
                value, null, NullRect);
      }
    }
  }

  public override void ChangeValue(GoChangedEventArgs e, bool undo) {
    switch (e.SubHint) {
      case ChangedMaxLinks:
        this.MaxLinks = e.GetInt(undo);
        return;
      default:
        base.ChangeValue(e, undo);
        return;
    }
  }

  // Event hints
  public const int ChangedMaxLinks = LastChangedHint + 11;

  // LimitedPort state
  private int myMaxLinks = 999999;
}
```

Handling Big Changes

Keeping track of all these edits is simple enough, but incurs a lot of overhead for detecting the change and remembering the **GoChangedEventArgs**. What should you do when you know you might be making a lot of changes and don't want the repeated overhead?

Your initial reaction might be to suspend updates. Setting **SuspendsUpdates** to true will turn off all event notification. After all of the batched changes are done,

you would set **SuspendsUpdates** to false to re-enable event notification, and event handlers would have to assume anything and everything had possibly changed. This is true both at the **GoDocument** level as well as the **GoObject** level.

Suspending updates is still possible, but with the introduction of undo managers, it is more complicated. The problem is that implementing undo requires getting the state *before* the changes. Turning off event notification means that there's no way to keep track of any changes that are going on. Trying to save all state at the time of the setting **SuspendsUpdates** true would be horribly inefficient, particularly for documents. Instead it is better to save very targeted state, depending on the kinds of changes that are expected to occur during the update suspension.

To accomplish this saving of state beforehand, Go supports a mechanism analogous to the **GoDocument.RaiseChanged** and **GoObject.Changed** mechanism used for notification after a change. The **RaiseChanging**/**Changing** methods are exactly like the **RaiseChanged**/**Changed** methods except they should be called just before a change. The changing methods don't need any old or new value parameters because the old state of the object is still current and the future state is not yet known.

Here is an example of how before changing can be done. This is how the **GoDocument.AllArranged** case is implemented, when trying to modify the locations of all of the nodes (and links) in a document:

```
// care about undo/redo, so need to call RaiseChanging here,
// so that the before-layout geometries of all top-level
// objects can be remembered
RaiseChanging(AllArranged, 0, null);
this.SuspendsUpdates = true;
LayoutWholeDiagram();
this.SuspendsUpdates = false;
// care about undo/redo, so need to call RaiseChanged here;
// don't need to pass previous arrangement here
RaiseChanged(AllArranged, 0, null, 0, null, NullRect,
                                  0, null, NullRect);
```

The "Changing" methods create **GoChangedEventArgs**s whose **IsBeforeChanging** property is true. Event handlers that don't care about notification before a change should ignore these events; for example, **GoView** ignores these events. But **GoUndoManager**, described below, uses them to remember the state before the event.

The **CopyOldValueForUndo** method, when invoked for the `AllArranged` **Changed** event, is responsible for getting the old/previous state. Since that state is not passed in via the previous value parameters, it must copy it from the current state of the document. Here we assume the

> CopyAllNodeLocationsAndLinkPaths method produces an ArrayList holding all the PointF information about the nodes and links.

```
public override void CopyOldValueForUndo(GoChangedEventArgs e) {
   switch (e.Hint) {
     . . .
     case AllArranged:
       // There's no previous value info passed in, must produce it
       if (e.IsBeforeChanging) {  // only beforehand, for undo
         e.OldValue = CopyAllNodeLocationsAndLinkPaths();
       }
       return;
     default: base.CopyOldValueForUndo(e); return;
   }
}
```

> The **CopyNewValueForRedo** method is implemented in a similar manner for the AllArranged case.

```
public override void CopyNewValueForRedo(GoChangedEventArgs e) {
   switch (e.Hint) {
     . . .
     case AllArranged: {
       // There's no new value info passed in, must produce it
       if (!e.IsBeforeChanging) {  // only afterwards, for redo
         e.NewValue = CopyAllNodeLocationsAndLinkPaths();
       }
       return; }
     default: base.CopyNewValueForRedo(e); return;
   }
}
```

> The **ChangeValue** method, called during undo and redo, sets all the node and link geometries given the information in the array stored in the **GoChangedEventArgs.**

```
public override void ChangeValue(GoChangedEventArgs e, bool undo) {
   switch (e.Hint) {
     . . .
     case AllArranged: {
       // Move the nodes and links according to info in saved array
       ArrayList copy = (ArrayList)e.GetValue(undo);
       RestoreNodeLocationsAndLinkPaths(copy);
       InvalidateViews();
       return; }
     default: base.ChangeValue(e, undo); return;
   }
}
```

GoUndoManager, CompoundEdits and Transactions

The edits implemented by **GoChangedEventArgs** are very detailed, specific changes that can be undone and redone. But when a user drags a selection, the user is changing the positions of possibly thousands of objects. Clearly the user will not expect that an undo command only move one of those objects back to its earlier location.

Go implements a **CompoundEdit** class for keeping track of an ordered list of **IGoUndoableEdit**s. Each compound edit is composed of all the edits that occur due to a particular user gesture or command. The compound edits in turn are managed by the undo manager.

GoUndoManager is a **GoDocument.Changed** event handler so that it can detect all of the changes that happen to a document, and then record them by producing and collecting **GoChangedEventArg**s in the **GoUndoManager**'s current **CompoundEdit**.

To control when a compound edit is started and finished, **GoUndoManager**s support the notion of a transaction. Call **StartTransaction** before any changes occur and call **FinishTransaction** afterwards. The first detected document change will open up a new compound edit. All succeeding edits are added to this current compound edit. A call to **FinishTransaction** will close up the current compound edit and add it to the undo manager's list of undoable edits.

For convenience, **StartTransaction**, **FinishTransaction**, and **AbortTransaction** are defined on **GoDocument** to call the corresponding method on the document's undo manager. They are also defined on **GoView**, to call the corresponding method on the view's document.

Views and tools are naturally responsible for detecting the start of a user action or command and knowing when it is finished. Thus the default implementations of many commands in **GoView** and **GoTool** start and end transactions. These methods include:

- **EditCopy** (start and end)
- **EditCut** (start and end)
- **EditPaste** (start and end)
- **DoExternalDrop** (start and end)
- **MoveSelection** (start and end)
- **CopySelection** (start and end)
- **DeleteSelection** (start and end)
- **GoTool.Start** (start)

- **GoTool.Stop** (end)

In addition, some methods such as **GoText.DoBeginEdit** and **DoEndEdit** enclose editing activity within a transaction. However, any code anywhere can start and end transactions on a document. When you add your own commands to your application, you will probably want to wrap any document changing code with a transaction.

Transactions may be nested (e.g. start, start, end, end). Only the final transaction end causes the compound edit to be closed and added to the undo manager's list. Beware calling **StartTransaction** without a corresponding call to **FinishTransaction**, perhaps due to an exception.

Each document that supports undo must have a **GoUndoManager**. Normally each document will have its own undo manager, but when there are interrelated documents where one change affects other documents, you may want to share one undo manager amongst several documents. Setting **GoDocument.UndoManager** automatically makes the manager a changed event handler on that document.

A call to **FinishTransaction** requires a **String** argument that describes that particular transaction to the user. This is the "presentation name". **GoUndoManager** provides default presentation names for the predefined transactions. These are the only strings in Go that should be localized for international applications. You can do the localization by setting **GoUndoManager.ResourceManager**, which **GoUndoManager.GetPresentationName** uses to try to replace the default presentation name for transactions.

A call of **AbortTransaction** will discard the current compound edit, rather than adding it to the undo manager. Unlike a transactional database system, aborting a transaction in Go does *not* automatically undo all of the changes that may have happened since the transaction start. This is because there might not be an undo manager, or because not all changes are being recorded.

Another difference between transactions with Go documents and database systems is that there is no prohibition on examining or even modifying documents or their objects without a preceding call to **StartTransaction**. There is no practical way to enforce the prohibition of reading the data structures.

Defining Menu Commands

GoUndoManager provides implementations of **Undo**, **Redo**, **CanUndo**, **CanRedo**, and **Clear** that user interface implementations should call.

GoDocument provides these same methods by delegating to the document's undo manager, if one exists. **GoView** also provides these same methods, by delegating to the document.

The following code is taken from the examples. Adding user-interface support for undo entails calling **CanUndo** to enable/disable the menu item and calling **Undo** to perform the action. In addition, you may wish to customize the menu item text with the presentation name.

```
editUndoMenuItem.Enabled = view.CanUndo();
if (editUndoMenuItem.Enabled) {
  editUndoMenuItem.Text = "Undo " +
    view.Document.UndoManager.UndoPresentationName;
} else {
  editUndoMenuItem.Text = "Undo";
}
editRedoMenuItem.Enabled = view.CanRedo();
if (editRedoMenuItem.Enabled) {
  editRedoMenuItem.Text = "Redo " +
    view.Document.UndoManager.RedoPresentationName;
} else {
  editRedoMenuItem.Text = "Redo";
}
```

8. XML AND SVG

GoDiagram provides support for Extensible Meta-Language (XML) and Scalable Vector Graphics (SVG) documents through the Northwoods.Go.Xml and Northwoods.Go.Svg assemblies, respectively.

The Xml assembly allows the developer to both read and write XML documents corresponding to particular GoDiagrams. The format of these XML documents (the document type definition or schema) is entirely controlled by the developer.

The Svg assembly is a specific use of the Xml assembly that specifies how to generate SVG from a GoView. Note that reading SVG documents into GoDiagram is not supported.

The Northwoods.Go.Xml assembly consists of 4 principal classes:

- **GoXmlReader**
- **GoXmlWriter**
- **GoXmlTransformer**
- **GoXmlBindingTransformer**

The **GoXmlWriter** and **GoXmlReader** classes define the **Generate** and **Consume** methods that provide the basic control for the writing and reading of XML documents.

The **GoXmlTransformer** class provides the detailed information necessary to transform a particular GoDiagram class to or from an XML element. Both **GoXmlReader** and **GoXmlWriter** have a collection of **GoXmlTransformers** associated with them. A **GoXmlTransformer** subclass should be defined for each GoDiagram class that needs to be translated to or from XML.

The **GoXmlBindingTransformer** class inherits from **GoXmlTransformer** and provides a way to establish bindings between XML element attributes and object properties.

Writing and Reading XML

Some Simple Examples using GoXmlBindingTransformer

For the majority of simple diagrams, using **GoXmlBindingTransformer** should be sufficient to implement persistence using custom XML. This assumes that you can define your own application-specific XML schema – i.e. you do not need to meet the requirements of some predefined XML format.

Let us consider a diagram consisting of **GoBasicNode**s and **GoLabeledLink**s.
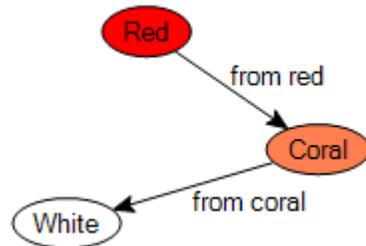
The XML content for this diagram in our application will look like:

```
<graph>
  <node Port="0" label="Red" color="-65536" loc="216 171" />
  <node Port="1" label="Coral" color="-32944" loc="302 230" />
  <node Port="2" label="White" color="-1" loc="176 267" />
  <link from="0" to="1" label="from red" />
  <link from="1" to="2" label="from coral" />
</graph>
```

There are three different kinds of XML elements: one for the document, one for the nodes, and one for the links. Each kind of element has a number of attributes that correspond to property values on their respective objects. We can declare bindings between XML element attributes and object properties by using the **GoXmlBindingTransformer** class. A transformer knows how to convert an XML element into a corresponding object and the reverse process. In addition, a **GoXmlBindingTransformer** has a **Prototype** property which is copied to create a new instance of an object when the XML element is read. It also has an **ElementName** property that specifies the name of the XML element during writing and during reading, the latter to identify which transformer to use.

We need to define three **GoXmlBindingTransformer**s, one for each kind of XML element. We will put this code into a single method that can be used for initializing both a **GoXmlWriter** as well as a **GoXmlReader**, since the bindings should be the same when writing as well as when reading XML.

For each kind of object we need to create an instance of it that can be copied. You can do as much initialization as you want for each prototype object. Each transformer is registered with the reader or writer by calling **AddTransformer**.

The arguments to the **GoXmlBindingTransform** constructor are the **ElementName** and the **Prototype** object. If the element name is not supplied, the name of the **Type** of the prototype object is used.

The calls to **AddBinding** establish the mapping between an attribute name and a property for that particular transformer. The property specifier is often a simple property name, but it can be a "path" of property names separated by periods. This is useful when the value you want to get is not a property on the immediate object, but is a property on a related object, typically on a part of it.

```
public void RegisterTransformers(GoXmlReaderWriterBase rw) {
  // create a prototype document
  GoDocument doc = new GoDocument();
  GoXmlBindingTransformer bt = new GoXmlBindingTransformer("graph", doc);
  rw.AddTransformer(bt);

  // create a prototype node
  GoBasicNode bn = new GoBasicNode();
  bn.LabelSpot = GoObject.Middle;
  bn.Text = "";
  GoXmlBindingTransformer bt1 = new GoXmlBindingTransformer("node", bn);
  // generates attributes for all named ports, to define their id's
  // without generating separate elements for them
  bt1.HandlesNamedPorts = true;
  // map the "label" attribute to the GoBasicNode.Text property
  bt1.AddBinding("label", "Text");
  // the "color" attribute is the GoBasicNode.Shape.BrushColor property
  bt1.AddBinding("color", "Shape.BrushColor");
  bt1.AddBinding("loc", "Location");
  rw.AddTransformer(bt1);

  // create a prototype link
  GoLabeledLink ll = new GoLabeledLink();
  ll.ToArrow = true;
  GoText lab = new GoText();
  lab.Selectable = false;
  ll.MidLabel = lab;
  GoXmlBindingTransformer bt2 = new GoXmlBindingTransformer("link", ll);
  // the "from" attribute will be a reference to the
  // GoLabeledLink.FromPort object
  bt2.AddBinding("from", "FromPort");
  bt2.AddBinding("to", "ToPort");
  // the "label" value is the GoLabeledLink.MidLabel.Text property
  bt2.AddBinding("label", "MidLabel.Text");
  rw.AddTransformer(bt2);
}
```

We also need to handle references between objects. By far the most common case of this is where a link refers to two ports. If you set the **HandlesNamedPorts** property to true on a **GoXmlBindingTransformer** for a node class, it will automatically read and write XML attributes for each **GoPort** that is a named child of the node. The name of the port is used as the attribute name. The attribute value is a unique identifier. This identifier can be used as

the value for an attribute that corresponds to a property that refers to the port. In this case, **GoBasicNode** just has a single port, named "Port".

Writing an XML file is just:

```
GoXmlWriter xw = new GoXmlWriter();
RegisterTransformers(xw);
xw.Objects = goView1.Document;
using (StreamWriter file = new StreamWriter(@"C:\temp\test.xml")) {
  xw.Generate(file);
}
```

Reading an XML file into a newly created **GoDocument** is:

```
GoXmlReader xr = new GoXmlReader();
RegisterTransformers(xr);
using (StreamReader file = new StreamReader(@"C:\temp\test.xml")) {
  goView1.Document = (GoDocument)xr.Consume(file);
}
```

Note how the shared **RegisterTransformers** method is used to make sure the same transformers are defined when writing as when reading.

Of course you can easily customize what information is stored in the XML by adding more attribute/property bindings.

You can handle other object types fairly easily. For example, **GoIconicNode:**

```
GoIconicNode ic = new GoIconicNode();
ic.Initialize(null, "", "");
bt = new GoXmlBindingTransformer("iconicnode", ic);
bt.HandlesNamedPorts = true;
bt.AddBinding("name", "Image.Name");
bt.AddBinding("index", "Image.Index");
bt.AddBinding("iconsize", "Icon.Size");
bt.AddBinding("loc", "Location");
rw.AddTransformer(bt);
```

Or **GoGeneralNode:**

```
GoGeneralNode gn = new GoGeneralNode();
gn.Initialize(null, "", "", "", 0, 0);
bt = new GoXmlBindingTransformer(gn);
// each node gets a unique id
bt.IdAttributeUsedForSharedObjects = true;
// each port gets a separate child element
bt.GeneratesPortsAsChildElements = true;
bt.BodyConsumesChildElements = true;
bt.AddBinding("name", "Image.Name");
bt.AddBinding("index", "Image.Index");
bt.AddBinding("iconsize", "Icon.Size");
bt.AddBinding("top", "TopLabel.Text");
bt.AddBinding("bottom", "BottomLabel.Text");
```

```
bt.AddBinding("loc", "Location");
rw.AddTransformer(bt);
```

Note that for **GoGeneralNode** there may be an arbitrary number of ports, each with its own name and need for unique identifier. By setting the **GeneratesPortsAsChildElements** property, the transformer will generate an element for each port. We also set **BodyConsumesChildElements** to make sure that when reading XML, it will try to create objects for each of the nested XML elements.

That means we also need to define a transformer for the **GoGeneralNodePort** class:

```
GoGeneralNodePort gnp = gn.MakePort(true);
bt = new GoXmlBindingTransformer(gnp);
// each port gets a unique id
bt.IdAttributeUsedForSharedObjects = true;
bt.AddBinding("left", "LeftSide");
bt.AddBinding("name", "Name");
rw.AddTransformer(bt);
```

Setting the **IdAttributeUsedForSharedObjects** property to true is necessary to make sure the "id" attribute is used to record a unique identifier for each port, unique across the whole document. The two transformers together produce XML such as:

```
<GoGeneralNode id="20" name="star.gif" index="-1"
    iconsize="20 24.14583" top="top" bottom="bottom" loc="71 275">
  <GoGeneralNodePort id="16" left="true" name="L0" />
  <GoGeneralNodePort id="17" left="true" name="L1" />
  <GoGeneralNodePort id="18" left="false" name="R0" />
  <GoGeneralNodePort id="19" left="false" name="R1" />
</GoGeneralNode>
```

Although there is a limit to the complexity of diagrams that you can read and writing using **GoXmlBindingTransformer** without having to override any methods of **GoXmlTransformer**, you may be able to handle **GoSubGraph** and other more complex classes.

The transformer for **GoSubGraph** depends on the **HandlesChildren** property to cause the transformer to generate nested XML elements in the body of the element for the subgraph. When consumed, those nested elements result in **GoObject**s that are added to the subgraph.

```
GoSubGraph sg = new GoSubGraph();
sg.Port = new GoPort();  // make each subgraph have a Port
sg.Port.FromSpot = GoObject.NoSpot;
sg.Port.ToSpot = GoObject.NoSpot;
bt = new GoXmlBindingTransformer("GoSubGraph", sg);
// to generate id for GoSubGraph as a node
```

```
bt.IdAttributeUsedForSharedObjects = true;
// to generate id for GoSubGraph.Port
bt.HandlesNamedPorts = true;
// generates children and consumes them by adding to the subgraph
bt.HandlesChildren = true;
// make sure reading/writing each child calls the
// Generate/ConsumeChildAttributes methods
bt.HandlesChildAttributes = true;
// add attributes for SavedBounds or SavedPath to each child node
// or link when the subgraph is collapsed
bt.HandlesSubGraphCollapsedChildren = true;
bt.AddBinding("back", "BackgroundColor");
bt.AddBinding("opacity", "Opacity");
bt.AddBinding("border", "BorderPen.Color");
bt.AddBinding("borderwidth", "BorderPen.Width");
bt.AddBinding("loc", "Location");
// define these AFTER defining Location binding
bt.AddBinding("wasexpanded", "WasExpanded");
bt.AddBinding("expanded", "IsExpanded");
rw.AddTransformer(bt);
```

The **GoSubGraph** transformer also depends on the **HandlesChildAttributes** and **HandlesSubGraphCollapsedChildren** properties, which are responsible for making sure each nested child element gets additional attributes specified by the (parent) subgraph. This information is used to associate saved information for each child node and link when the subgraph is collapsed.

Here are the transformers for the example class **ClassDiagramNode**. There are three transformers because there are logically three levels of nesting of **GoObject**s in each **ClassDiagramNode**.

```
ClassDiagramNode cdn = new ClassDiagramNode();
bt = new GoXmlBindingTransformer(cdn);
bt.IdAttributeUsedForSharedObjects = true;
bt.HandlesNamedPorts = true;
bt.HandlesChildren = true;  // generates and consumes child objects
// collection of children is held in this property:
bt.ChildrenCollectionPath = "MyBody";
bt.AddBinding("spread", "LinkPointsSpread");
bt.AddBinding("loc", "Location");
bt.AddBinding("startcolor", "StartColor");
bt.AddBinding("endcolor", "EndColor");
bt.AddBinding("desc", "Description.Text");
bt.AddBinding("itemwidth", "ItemWidth");
rw.AddTransformer(bt);

ClassDiagramNodeItemList cdnil = cdn.MakeList("");
bt = new GoXmlBindingTransformer(cdnil);
bt.HandlesChildren = true;  // generates and consumes child objects
// collection of children is held in this property:
```

```
bt.ChildrenCollectionPath = "List";
bt.AddBinding("name", "Text");
bt.AddBinding("itemwidth", "ItemWidth");
rw.AddTransformer(bt);

ClassDiagramNodeItem cdni = cdn.MakeItem("", "");
bt = new GoXmlBindingTransformer(cdni);
bt.AddBinding("text", "Text");
bt.AddBinding("img", "Image.Name");
bt.AddBinding("imgidx", "Image.Index");
rw.AddTransformer(bt);
```

The **ChildrenCollectionPath** property specifies a property of the object that is supposed to be the collection of objects represented by the nested XML elements. When **ChildrenCollectionPath** is the empty string (which is the default), the child objects are taken from and are added to the object, which is assumed to be a **GoGroup**. In the case of **ClassDiagramNode** there are additional layers of groups that are not reflected in the logical nesting of XML elements.

Tree Structured Graphs

When your graphs consist of nodes and links connected in a tree-like fashion, you might not want to represent each link as a separate XML element, since you can specify the tree-parent node by using an XML attribute. The XML data is basically just a list of nodes:

```
<graph>
  <node id="0" label="Root" />
  <node id="1" label="A1" parent="0" />
  <node id="2" label="A2" parent="0" />
  <node id="3" label="B1" parent="2" />
  <node id="4" label="C1" parent="3" />
  <node id="5" label="D1" parent="4" />
  <node id="6" label="D2" parent="4" />
  <node id="7" label="B2" parent="2" />
</graph>
```

A single transformer is needed:

```
GoBasicNode n = new GoBasicNode();
n.LabelSpot = GoObject.Middle;
n.Text = "";
n.Shape = new GoRoundedRectangle();
GoXmlBindingTransformer tr = new GoXmlBindingTransformer("node", n);
// make sure each node gets a unique ID
tr.IdAttributeUsedForSharedObjects = true;
// provide the prototype link for connecting the nodes
tr.TreeLinkPrototype = new GoLink();
// indicate the direction of the link (from parent to child)
tr.TreeLinksToChildren = true;
```

```
tr.AddBinding("label", "Text");
// add an attribute that refers to the parent node in the tree
tr.AddBinding("parent", "TreeParentNode");
```

To write out this graph:

```
GoXmlWriter wrt = new GoXmlWriter();
wrt.AddTransformer(tr);
wrt.Objects = goView1.Document;
using (StreamWriter file = new StreamWriter( . . . )) {
  wrt.Generate(file);
}
```

To read and layout this graph:

```
GoXmlReader rdr = new GoXmlReader();
rdr.AddTransformer(tr);
rdr.RootObject = goView1.Document;
using (StreamReader file = new StreamReader( . . . )) {
  rdr.Consume(file);
}

GoLayoutTree layout = new GoLayoutTree();
layout.Document = doc;
// other customizations are described in the GoLayout User Guide
layout.PerformLayout();
```

This results in the following view:
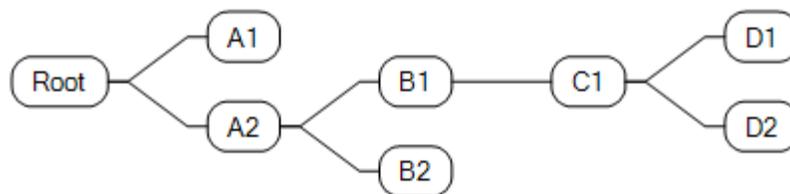
Tree Structured XML

Another common method of representing trees in XML is with the nesting of XML elements.

```xml
<graph>
  <node label="Root">
    <node label="A1" />
    <node label="A2">
      <node label="B1">
        <node label="C1">
          <node label="D1" />
          <node label="D2" />
        </node>
      </node>
      <node label="B2" />
```

```xml
      </node>
  </node>
</graph>
```

GoXmlBindingTransformer makes this easier by providing several "Tree…" properties.

```csharp
GoBasicNode n = new GoBasicNode();
n.LabelSpot = GoObject.Middle;
n.Text = "";
n.Shape = new GoRoundedRectangle();
GoXmlBindingTransformer tr = new GoXmlBindingTransformer("node", n);
// indicate that the XML consists of nested elements
tr.TreeStructured = true;
// provide the prototype link for connecting the nodes
tr.TreeLinkPrototype = new GoLink();
// indicate the direction of the link (from parent to child)
tr.TreeLinksToChildren = true;
tr.AddBinding("label", "Text");
```

Unlike the cases above, we need to give the **GoXmlWriter** the specific root node(s) from which to generate XML. If we used the whole document, each node would be written out separately, causing much duplication of information in subtrees. To write out this graph:

```csharp
GoXmlWriter wrt = new GoXmlWriter();
wrt.AddTransformer(tr);
// need to select root node(s)
GoCollection coll = new GoCollection();
coll.Add(goView1.Document.FindNode("Root"));
wrt.Objects = coll;
using (StreamWriter file = new StreamWriter( . . . )) {
  wrt.Generate(file);
}
```

To read and layout this graph:

```csharp
GoXmlReader rdr = new GoXmlReader();
rdr.AddTransformer(tr);
rdr.RootObject = goView1.Document;
using (StreamReader file = new StreamReader( . . . )) {
  rdr.Consume(file);
}

GoLayoutTree layout = new GoLayoutTree();
layout.Document = doc;
// other customizations are described in the GoLayout User Guide
layout.PerformLayout();
```

Writing XML Using GoXmlTransformer

Let's start by taking an overview of the process of generating an XML document from GoDiagram. The **GoXmlWriter** class has a **RootElementName** property that defines the name of the root XML element. The **GoXmlWriter** class also has an **Objects** property that defines the collection of objects that will be written out under this root element.

The **GoXmlWriter Generate** method iterates across this collection of objects in two passes. The first pass allows a table of shared objects to be created by calling the **GenerateDefinitions** method on the transformer associated with each object. The second pass actually generates the XML elements and their attributes by calling **GenerateElements**, **GenerateAttributes**, and **GenerateBody** on the transformer associated with each object. We will discuss these transformer methods in a few paragraphs.

A simple example is provided in the Processor sample application. The following code shows the **ProcessDocument.Store** method used to generate XML from a Process Flow Diagram:

```
public void Store(StreamWriter file) {
  GoXmlWriter writer = new GoXmlWriter();
  writer.RootElementName = "process";
  writer.AddTransformer(new TransformActivityNode());
  writer.AddTransformer(new TransformActivityPort());
  writer.AddTransformer(new TransformFlowLink());
  writer.AddTransformer(new TransformComment());
  writer.AddTransformer(new TransformRemoteConnectorNode());
  writer.Objects = this;
  writer.Generate(file);
}
```

The code above simply identifies the name of the root XML element, adds a **GoXmlTransformer** for each separate class that needs to be converted to XML, and supplies the document as the collection of **Objects** to be transformed into XML.

Clearly, the majority of the transformation process is occurring in the **GoXmlTransformer** subclasses, so we will next examine **GoXmlTransformer**s in more detail.

A **GoXmlTransformer** has a **TransformerType** property that specifies the class of objects that the transformer is associated with. It also has an **ElementName** property that specifies the XML element name to be generated for objects of that type. A **GoXmlTransformer** also has methods for the object definition and generation of the element and attributes associated with that type. As described earlier, the **GenerateDefinitions** method of the **GoXmlTransformer** will be invoked by the **GoXmlWriter.Generate** method on the first pass through the objects supplied by the **GoXmlWriter.Objects** property. The

GenerateElement, **GenerateAttributes**, and **GenerateBody** methods will be called on the second pass through the objects.

The behavior of the **GoXmlTransformer.GenerateDefinitions** method (assuming the **IdAttributeUsedForSharedObjects** property is true) is to simply add the object instance to an internal shared objects table (if not already in that table) and return an identifier uniquely associated with this object. Furthermore, the default behavior of the **GoXmlTransformer.GenerateAttributes** method is to add an "id" attribute with that identifier value to the element generated by the transformer. This allows us to refer to specific object instances from other objects within the generated XML by specifying these identifiers. We define this entire shared objects table during pass 1 so that we will have access to these identifiers for all the objects before we begin generating the actual XML elements. Typically, you do not need to override **GenerateDefinitions** unless the object associated with your **GoXmlTransformer** contains other objects that you wish to refer to in the generated XML, in which case you may want to call **Writer.DefineObject** for each contained child object, which will in turn cause **GenerateDefinitions** to be called on the transformer associated with those children.

The **GenerateElement** method is very simple and rarely needs to be overridden. The default behavior of **GenerateElement** is to generate an XML element with the name defined by the **GoXmlTransformer ElementName** property. The **ElementName** property is typically set in the constructor for your subclass of **GoXmlTransformer**.

The **GenerateAttributes** method allows you to specify the name and value of whatever attributes you would like to add to the element generated by **GenerateElement**. Attributes are added with the **WriteAttrVal** method. The **WriteAttrVal** method is overloaded by datatype of the attribute argument to convert attribute values of different data types to the String data type which is actually written as the attribute value.

The **GenerateBody** method allows you to generate any nested elements that are part of the rendering of an object. You may want to call **Writer.GenerateObject** for each contained child object, which will in turn cause **GenerateElement**, **GenerateAttributes**, and **GenerateBody** to be called on the transformer associated with those children.

Let's look at that portion of **TransformActivityNode** in the Processor sample application.that is used to write XML.

```
public class TransformActivityNode : GoXmlTransformer {
  public TransformActivityNode() {
    this.TransformerType = typeof(ActivityNode);
    this.ElementName = "activity";
    this.IdAttributeUsedForSharedObjects = true;
  }
```

```csharp
public override void GenerateDefinitions(Object obj) {
  base.GenerateDefinitions(obj);
  ActivityNode n = (ActivityNode)obj;
  foreach (IGoPort p in n.Ports) {
    this.Writer.DefineObject(p.GoObject);
  }
}
public override void GenerateAttributes(Object obj) {
  base.GenerateAttributes(obj);
  ActivityNode n = (ActivityNode)obj;
  WriteAttrVal("type", (int)n.ActivityType);
  WriteAttrVal("xy", n.Icon.Position);
  WriteAttrVal("size", n.Icon.Size);
  WriteAttrVal("label", n.Text);
  if (n.LabelOffset != new SizeF(-99999, -99999))
    WriteAttrVal("labeloffset", n.LabelOffset);
}
public override void GenerateBody(Object obj) {
  base.GenerateBody(obj);
  ActivityNode n = (ActivityNode)obj;
  foreach (IGoPort p in n.Ports) {
    this.Writer.GenerateObject(p.GoObject);
  }
}
}
```

The constructor in the code above specifies the type associated with the transformer and the name of the element to be generated for objects of that type. In addition, it sets **IdAttributeUsedForSharedObjects** to true to enable the recording and generation of object ids in the **GenerateDefinitions** phase of the process.

GenerateDefinitions simply calls the **Writer**'s **DefineObjects** method for the port objects contained in the **ActivityNode**. This causes the id attributes to be generated for these ports so they can be referred to by other elements in the generated XML (for example by **FlowLink**s).

GenerateAttributes specifies the attribute names and values for those things that can be different between different instances of ActivityNode in the Processor application. Note that this only writes out information that is needed for correct operation of the application. It does not write out incidental information, such as the font used by the node's **Label**, to allow flexibility on the part of the application to decide how to display the text.

GenerateBody simply calls the **Writer**'s **GenerateObjects** method for the port objects contained in the **ActivityNode**. This causes nested "port" elements to be generated within the "activity" element–the details are defined by the **TransformActivityPort** class.

The XML output by this code for a simple Process Flow Diagram consisting of 2 nodes and 1 link is as follows:

```xml
<process>
  <activity id="0" type="0" xy="131 124" size="48 48" label="Start">
    <port id="1" UserFlags="0" xy="171 144" spot="64" />
  </activity>
  <activity id="2" type="1" xy="244 124" size="48 48" label="Finish">
    <port id="3" UserFlags="0" xy="244 144" spot="256" />
  </activity>
  <flow from="1" to="3"
        points="179 148 189 148 189 148 189 148 234 148 244 148"
        label="label" labeloffset="0 0" labelsegment="3"
        labelpercentage="50" />
</process>
```

We can see that the root element is indeed named "process", as specified in the **Store** method. We can see the "activity" elements and their attributes generated as specified by the **TransformActivityNode** class, and also the "port" elements generated by the **TransformActivityPort** class. Finally, we can see the "flow" element with attributes specifying the ids of the "from" and "to" ports of the link.

While we have discussed how id attributes are generated, we have not yet discussed how references to these ids are generated. We'll examine generating references to other object ids by examining the generation of the "to" and "from" attributes within the "flow" element in **TransformFlowLink**.

The **GenerateAttributes** method within this the **TransformFlowLink** class generates the attributes that refer to the port ids by using the **Writer**'s **FindShared** method. This method returns the id of an object, assuming that **IdAttributeUsedForSharedObjects** is true and that **GenerateDefinitions** has been called on that object in pass 1.

The following code fragment from the **GenerateAttributes** method of **theTransformFlowLink** demonstrates the use of the **FindShared** method and the generation of the "from" and "to" attributes.

```csharp
public override void GenerateAttributes(Object obj) {
  base.GenerateAttributes(obj);
  FlowLink flow = (FlowLink)obj;
  GoPort p = flow.FromPort as GoPort;
  if (p != null) {
    String fromid = this.Writer.FindShared(p);
    WriteAttrVal("from", fromid);
  }
  p = flow.ToPort as GoPort;
  if (p != null) {
    String toid = this.Writer.FindShared(p);
    WriteAttrVal("to", toid);
  }
  ...
```

```
      }
```

Reading XML Using GoXmlTransformer

Let's now examine the process of consuming an XML document to create **GoObject**s. The **GoXmlReader.Consume** method takes a file or **System.Xml.XmlDocument** as an argument and causes **Object**s to be created corresponding to the elements in the XML document. The **RootObject** property defines the list to which the newly created objects are added. If the **RootObject** is an **IGoCollection** such as a **GoDocument** and the newly created object is a **GoObject**, the newly created **GoObjects** are added to the **GoDocument** and are immediately visible in any **GoView** for that **GoDocument**.

The **GoXmlReader.Consume** method calls **ConsumeRootElement**, **ConsumeRootAttributes**, **ConsumeRootBody**, and **ProcessDelayedObjects**. If your root element has attributes you can override **ConsumeRootAttributes** to read those attributes. **ConsumeRootBody** will call **ConsumeObject** on each of the elements directly contained in the root element of the XML file. This will in turn call the **Allocate**, **ConsumeAttributes**, and **ConsumeBody** methods on the **GoXmlTransformer** associated with the element name. **ProcessDelayedObject** updates references to objects that may have been generated as the new objects were created.   Note that **ProcessDelayedObject** runs only after the all the elements have been processed and all the objects corresponding to these elements have been created. We will say more about this process in the following paragraphs.

A simple example is provided in the Processor sample application. The following code shows the **ProcessDocument.Load** method used to read an XML document and create a Process Flow Diagram:

```
public void Load(StreamWriter file) {
  StartTransaction();
  Clear();
  GoXmlReader reader = new GoXmlReader();
  reader.AddTransformer(new TransformActivityNode());
  reader.AddTransformer(new TransformActivityPort());
  reader.AddTransformer(new TransformFlowLink());
  reader.AddTransformer(new TransformComment());
  reader.AddTransformer(new TransformRemoteConnectorNode());
  reader.RootObject = this;
  reader.Consume(file);
  FinishTransaction("loaded from file");
}
```

The code above simply adds a **GoXmlTransformer** for each separate class that needs to be created from the elements in the XML and indicates the **RootObject** to which the newly created objects should be added. The **RootObject** is the

ProcessDocument (a subclass of **GoDocument**), so the **GoObjects** created from the elements will simply be added to this **GoDocument**.

Once again, the majority of the transformation process is occurring in the **GoXmlTransformer** subclasses, so we will next examine those **GoXmlTransformer**s in more detail.

A **GoXmlTransformer** has an **ElementName** property that specifies the XML element associated with the transformer. It also has a **TranformerType** property that specifies the type of object that will be created corresponding to that element. A **GoXmlTransformer** also has methods for the creation and initialization of the objects associated with an element. As described earlier, the **Allocate**, **ConsumeAttributes**, and **ConsumeBody** methods will be invoked for each element directly contained by the root element of the XML document.

The default behavior of the **GoXmlTransformer.Allocate** method is to create an instance of the class given by **TransformerType**. If you require additional initialization not provided by the default (zero-argument) constructor for this class you can override this method.

The **ConsumeAttributes** method reads the values of the element attributes. If the **IdAttributeUsedForSharedObjects** property is true, **GoXmlTransformer ConsumeAttributes** will also call the **Reader**'s **MakeShared** method to register the "id" attribute of the object with the actual object instance it corresponds to. Objects that have been created can then be looked up by calling the **GoXmlReader.FindShared** method with the object id of the object to be returned.

 GoXmlTransformer has a number of methods that are typically used within **ConsumeAttributes** to read attribtutes of different datatypes. These methods include:

- **StringAttr**

- **Int32Attr**

- **SingleAttr**

- **BooleanAttr**

- **PointFAttr**

- **SizeFAttr**

- **RectangleFAttr**

- **ColorAttr**

- **TypeAttr**

- **Int32ArrayAttr**

- **SingleArrayAttr**

- **PointFArrayAttr**

- **ColorArrayAttr**

These methods correspond to the datatypes that can be generated from the **WriteAttrVal** method.

If the **BodyConsumesChildElements** property is true, the **ConsumeBody** method will iterate through all the child elements of this element and call **ConsumeObject** on each child element, which will in turn call **Allocate**, **ConsumeAttributes**, and **ConsumeBody** on the **GoXmlTransformer** associated with each of these child elements. The object returned by **ConsumeObject** is then passed to **ConsumeChild**.

Let's look at that portion of **TransformActivityNode** in the Processor sample application.that is used to read XML.

```
public class TransformActivityNode : GoXmlTransformer {
  public TransformActivityNode() {
    this.TransformerType = typeof(ActivityNode);
    this.ElementName = "activity";
    this.BodyConsumesChildElements = true;
  }
  public override Object Allocate() {
    ActivityNode n = new ActivityNode();
    n.Initialize(null, "doc.gif", "");
    return n;
  }
  public override void ConsumeAttributes(Object obj) {
    base.ConsumeAttributes(obj);
    ActivityNode n = (ActivityNode)obj;
    n.ActivityType = (ActivityType)Int32Attr("type",
(int)n.ActivityType);
    if (n.ActivityType == ActivityType.Start ||
        n.ActivityType == ActivityType.Finish) {
      n.Image.Name = "star.gif";
    }
    n.Icon.Position = PointFAttr("xy", new PointF(100, 100));
    if (IsAttrPresent("size"))
      n.Icon.Size = SizeFAttr("size", n.Icon.Size);
    n.Text =  StringAttr("label", n.Text);
    if (IsAttrPresent("labeloffset"))
      n.LabelOffset = SizeFAttr("labeloffset", n.LabelOffset);
  }
  public override void ConsumeChild(Object parent, Object child) {
    base.ConsumeChild(parent, child);
    ActivityNode n = (ActivityNode)parent;
    n.Add(child);
  }
}
```

The constructor in the code above specifies the element name associated with the transformer and type of object to be created to correspond to those elements. In addition, it sets **BodyConsumesChildElements** to true to enable the automatic processing of child elements.

Allocate simply calls the constructor for ActivityNode and initializes that node by passing the "doc.gif" file to the **Initialize** method of **ActivityNode**.

ConsumeAttributes reads the "type", "xy", and "size" attributes of the "activity" element and sets the appropriate property values in the corresponding **ActivityNode**.

ConsumeChild causes each of the child objects of the ActivityNode to be added to the ActivityNode after they are created and initialized.

Finally, we need to examine how references to other objects (other than child objects) are created as elements are read. Once again, we'll look at the **TransformFlowLink** object to see how the link references to port ids are transformed into actual object references.

The following is a portion of the **ConsumeAttributes** method of **TransformFlowLink**.

```
public override void ConsumeAttributes(Object obj) {
  base.ConsumeAttributes(obj);
  FlowLink flow = (FlowLink)obj;
  String fromid = StringAttr("from", null);
  if (fromid != null) {
    GoPort from = this.Reader.FindShared(fromid) as GoPort;
    flow.FromPort = from;
  }
  String toid = StringAttr("to", null);
  if (toid != null) {
    GoPort to = this.Reader.FindShared(toid) as GoPort;
    flow.ToPort = to;
  }
        ...
}
```

To find the object that corresponds to the "from" or "to" port in the above code, we simply call the **GoXmlReader.FindShared** method passing the port id. The port instance corresponding to that id is looked up and returned.

But this assumes that the object being looked up already has been created and entered in the shared objects table. Why are these assumptions safe? In this case, we can assume that the object being looked up has already been created because it is a port object which is the child of a node. The **GoXmlWriter NodesGeneratedFirst** property is true by default. Thus **GoNode** objects (and their children) are generated before any links are created, so we can assume when

reading in the **FlowLink** from the XML that the ports being referenced have already been created. Furthermore, the referenced objects and their ids have already been entered into the table because of the behavior of **GoXmlTransformer.ConsumeAttributes** when **IdAttributeUsedForSharedObjects** is true. In this case, **ConsumeAttributes** will call **GoXmlWriter.MakeShared** with both the id attribute and value and the newly created object instance, which enters the object instance and its id in the table.

But what can we do if we don't know if the object being referenced has already been created? In this case we must call the **GoXmlReader.AddDelayedRef** method in our implementation of **GoXmlTransformer.ConsumeAttributes**. This will ultimately cause the **UpdateReference** method to be called in our **GoXmlTransformer** class, passing the attribute name, and object instance for the referred object. The **UpdateReference** call will not occur until all the objects have been created.   An example of this can be found in the **TransormRemoteConnectorNode** class of the Processor sample application.

Writing SVG

The SVG assembly is a specific use of the XML assembly that supports the generation of SVG from a **GoView**.

The Northwoods.Go.Svg assembly consists principally of the **GoSvgWriter** class (a subclass of **GoXmlWriter**) and several **GoSvgGenerator** classes (subclasses of **GoXmlTransformer**).

Using the **GoSvgWriter** class is extremely easy. One can typically render an entire **GoView** and all the **GoObjects** it displays by writing just 3 lines of code:

```
// example code for generating SVG:
GoSvgWriter w = new GoSvgWriter();
w.View = GetCurrentGoView();
w.Generate(@"C:\Demo1.svg");
```

The above code simply creates an instance of the **GoSvgWriter**, sets the **GoSvgWriter.View** property, and calls the **Generate** method. The **GoSvgWriter** itself automatically adds all the standard **GoSvgGenerator** classes (subclasses of **GoXmlTransformer**) that are necessary to render the **GoView**, including the rendering of all the standard **GoObject**s that are displayed in that view.

The developer may need to add their own **GoSvgGenerator** classes if they have created new **GoObject** subclasses that contain drawing code by overriding **GoObject.Paint**. In this case, one typically creates a new subclass of **GoSvgGenerator** and overrides the **GenerateBody** method to generate SVG elements that correspond to the drawing code in your **GoObject**. Note that in

order to simplify the generation of SVG elements, the **GoSvgGenerator** class has several methods that are similar to those found in the **Graphics** class. For example, the **GoSvgGenerator.WritePolygon** method can be used in place of the **Graphics.DrawPolygon** method.

As an example, let's say you have defined a class where you have overridden the **Paint** method as follows:

```
public class TriangleTextNode : GoTextNode {
  . . .
  public override void Paint(Graphics g, GoView view) {
    base.Paint(g, view);
    RectangleF r = this.Bounds;
    PointF[] pts = new PointF[3];
    pts[0] = new PointF(r.X+3, r.Y+3);
    pts[1] = new PointF(r.X+13, r.Y+3);
    pts[2] = new PointF(r.X+8, r.Y+13);
    g.FillPolygon(Brushes.Yellow, pts);
    g.DrawPolygon(Pens.Black, pts);
  }
}
```

If you want to get the same results in the generated SVG, you could define a generator as follows:

```
public class GeneratorTriangleTextNode : GoSvgGenerator {
  public GeneratorTriangleTextNode() {
    this.TransformerType = typeof(TriangleTextNode);
  }

  public override void GenerateBody(Object obj) {
    base.GenerateBody(obj);
    TriangleTextNode ttn = (TriangleTextNode)obj;
    RectangleF r = ttn.Bounds;
    PointF[] pts = new PointF[3];
    pts[0] = new PointF(r.X+3, r.Y+3);
    pts[1] = new PointF(r.X+13, r.Y+3);
    pts[2] = new PointF(r.X+8, r.Y+13);
    WritePolygon(Pens.Black, Brushes.Yellow, pts);
  }
}
```

Note how the call to **base.GenerateBody** corresponds to a call to **base.Paint**, and how the call to **WritePolygon** corresponds to calls to **Graphics.FillPolygon** and **Graphics.DrawPolygon**.

To add this new **GoSvgTransformer**, the previous sample code would be modified as follows:

```
// example code for generating SVG:
```

```
GoSvgWriter w = new GoSvgWriter();
w.AddTransformer(new GeneratorTriangleTextNode());
w.View = GetCurrentGoView();
w.Generate(@"C:\Demo1.svg");
```

The SVG generated by **GoSvgWriter** not only defines a static image of the diagram, but also defines scripts and additional controls that can provide dynamic behavior as well. These include:

- Selection

- Tool Tips

- Hot Links (hrefs)

- Panning and Zooming

Of course, these additional behaviors can be turned on or off, customized, or entirely replaced by the developer. They are controlled by various properties and methods of the **GoSvgWriter** class.

The boolean **Scripting** property determines whether any scripts are generated at all. If this property is false, no scripts are generated and no dynamic behavior is possible.

The **ScriptFile** property determines what scripts are generated. By default, **ScriptFile** is an empty string, which causes GoDiagram's standard SVG JavaScripts to be generated. If this value is not empty, it is assumed to be a URL to a file containing the scripts that will be included by reference.

The boolean **ToolTips** property determines whether or not tool tips are displayed when the mouse hovers over an SVG element representing a **GoObject**. By default, **ToolTips** is true, so any **GoObject** that overrides the **GoObject.GetToolTip** method will display that tool tip in the generated SVG.

The **GetHref** method takes a **GoObject** argument and can be overridden to return a URI. When a user clicks on an SVG element that corresponds to such a **GoObject**, the resource associated with that object's URI is displayed by the SVG user agent.

The boolean **PanAndZoomControls** property determines whether or not a control is created in the generated SVG that the user can interact with to pan (scroll) or zoom (scale) the digram. By default, **PanAndZoomControls** is true. This control appears as follows:

By clicking on the points of the control the user can pan in that direction. By clicking on the + or - the user can zoom in or out. By clicking on the square in

the center, the user can return to the original zoom and pan values. By clicking on the numeric scale value, the scale is reset to 1.

Other more general script customizations can be easily accomplished by overriding the **GenerateScript** method. By calling the base **GenerateScript** method, you can cause all the standard GoDiagram SVG JavaScript to be generated. By calling **WriteStartElement**, **WriteTextBody**, and **WriteEndElement** you can then add your own script functions to extend these standard scripts. In particular, you can generate your own InitializeForms and or UpdateForms script functions. The InitializeForms script function will be called once to allow you to perform any initialization. The UpdateForms function script function will be called after a mouse up operation to allow you to perform an operations in response to user mouse click. For example, the following code will generate JavaScript that you can modify to do anything you want in response to a mouse click.

```
    protected override void GenerateScript() {
      if (!this.Scripting) return;
      base.GenerateScript();
      WriteStartElement("script");
      WriteTextBody(@"
function UpdateForms() {
  // find a selected object
  for (var id in goSelection) {
    var obj = goGetSelectable(id);
    // Do whatever you like with the selected objects--
    // the following displays the object id.
    alert('object id = ' + id);
  }
}
");
      WriteEndElement();
    }
  }
```

Note that the sample code above is writing JavaScript that is simply enclosed in a verbatim string literal. The UpdateForms() JavaScript function will be called by the standard GoDiagram JavaScript generated by the **base.GenerateScript()** call.

A more complex example of script customization can be found in the Demo1 sample application. In this sample a property sheet is displayed in response to clicking on an object. In your application you will probably need to make sure additional information is generated for each node so that your JavaScript code will be able to take the actions desired, such as displaying information for a node or invoking some action on a server.

9. PERFORMANCE HINTS

When there are only thousands of objects in a document, performance is rarely a problem. However, when dealing with many thousands of objects, the programmer should be aware of performance issues.

Usually the bottleneck in performance is GDI+ drawing speed. The use of partial transparency, linear gradients, or path gradients will definitely consume more resources and slow responsiveness. Dashed or dotted pens also cost drawing time, although not as badly as fancy brushes. The use of grids, particularly with small cell sizes, can impose a significant painting cost.

Use the simplest kind of **GoShape** that will serve your purpose. Although it may be very convenient to use a **GoDrawing** shape of a particular **GoFigure**, employing thousands of such shapes will consume both space and time.

Don't add an object, particularly complex objects such as groups, to the document until the last possible moment—as objects are modified or as objects are added to a group, no views or undo manager will be notified until after the object/group is added to the document.

If you need to change the bounds of an object, it is more efficient to change it once than to do so in several steps. For example, if you want to stretch the left edge of a rectangle further to the left while keeping the right edge at the same X position, you might do:

```
aRect.Left -= 20
aRect.Width += 20
```

This may get you the right result, but will involve two updates to the object, to its parent group, and to its document and views. Instead

```
aRect.Bounds = new RectangleF(aRect.Left-20, aRect.Top,
                              aRect.Width+20, aRect.Height)
```

will avoid the extra updates. Furthermore it is more likely to avoid problems with the layout of children in groups, because in the two-step procedure the **GoGroup.LayoutChildren** method might adjust everything based on the fact that the rectangle is (temporarily) no longer as far right as it used to be.

Speaking of **LayoutChildren**, that method can get called a lot. Each time any child's **Bounds** changes, or when a child is added or removed from the

GoGroup, will result in a call to **LayoutChildren**. When you have a node with hundreds of children, you may find it necessary to follow the convention that **LayoutChildren** do nothing when **GoObject.Initializing** is true. That will allow you to initialize or make wholesale changes to your group without performing any real work in your **LayoutChildren** override, until your code has set **Initializing** back to false and then calls **LayoutChildren(null)** explicitly to make sure everything is in its place.

Support for undo and redo slows down editing because the undo manager must listen for document events and construct edits for each change. By default a document does not have an undo manager, so you should set **GoDocument.UndoManager** only when needed.

Those undo edits can take up a lot of memory. Depending on your application design, sometimes you may wish to call **GoUndoManager.Clear** to save on virtual memory occupied by all of the edits. This is traditionally done when the document is saved, but you may implement your own policies. You can also change how much is saved by overriding **GoUndoManager.SkipEvent**.

If you want to limit the amount of memory consumed by the **GoUndoManager** due to the number of transactions that occur, you can set the **MaximumEditCount** property. This, however, does not limit the amount of memory used by each **CompoundEdit**.

Dragging performance can be slowed by setting **GoView.DragsRealtime** to true. However, if you have to have that property be true to get the desired interactive feedback during dragging, but you have a lot of links that are **Orthogonal** and **AvoidsNodes,** you can set **GoView.DragRoutesRealtime** to false. That will avoid the rerouting of links during the drag, only routing each link once at the end of the drag.

If you have **GoView.ObjectGotSelection** and **GoView.ObjectLostSelection** event handlers that are slow because they need to update other **Control**s, you may notice that selecting or deselecting a lot of objects takes a lot of time. We suggest that you additionally implement **GoView.SelectingStarting** and **GoView.SelectionFinished** event handlers. The former should disable updating; the latter should re-enable updating and make sure everything is up-to-date.

Reading and writing XML files can be very easy to implement when using **GoXmlBindingTransformer**. However, its use of reflection and its needing to interpret property paths does slow it down compared to the identical functionality implemented using a custom **GoXmlTransformer**.

10. GODIAGRAM 4.0: UPGRADING TO .NET 2.0 GENERICS AND COLLECTIONS

This section documents differences we found between **System.Collections** and **System.Collections.Generic**, since they can be somewhat confusing. You may find the information useful even if you are not working on **GoDiagram**.

This section also covers the design and coding changes involving this upgrade to require NET 2.0.

If you aren't thoroughly familiar with C# generics already, you can read:

http://msdn.microsoft.com/en-us/library/512aeb7t(VS.80).aspx

In the tables below, for each pair of non-generic and generic types, we list their corresponding methods and properties. On the left side is the non-generic type/method/property. We use "O" as a short hand for **Object**. On the right side is the generic type/method/property. "T", "K", "V" are generic type parameters. The table for **ICollection** also includes **IGoCollection** for comparison. There, "GO" is short hand for **GoObject**.

In each table we list all of the methods and properties, including inherited ones. Those that are inherited we prefix with the interface that they come from, just as you would when defining a class that implements the interface using explicit interface definitions.

IEnumerable and IEnumerable<T>

In the case of defining a class that implements **IEnumerable<T>**, this means implementing **IEnumerable.GetEnumerator()** as well as **IEnumerable<T>.GetEnumerator()**.

IEnumerable	IEnumerable<T> : IEnumerable
IEnumerator GetEnumerator()	IEnumerator IEnumerable.GetEnumerator()
	IEnumerator<T>

But sometimes there's more – you'll want to implement a specific **GetEnumerator()** for your enumerable class. That's the case in **GoDiagram**, where for efficiency reasons we defined types that were both **IEnumerable** and **IEnumerator** – for example, the**GoCollection** class uses the **GoCollectionEnumerator** struct that implements both. To make this struct accessible to the compiler for efficiency, one has to define **GetEnumerator()** to return the type **GoCollectionEnumerator**. Hence

```
public class GoCollection : ICollection<GoObject> {  // actually
IGoCollection, which inherits from ICollection<GoObject>

 // explicit interface definition for IEnumerable – cannot have access
modifiers

  IEnumerator IEnumerable.GetEnumerator() {

    return this.GetEnumerator();  // need "this." to disambiguate

  }

  // explicit interface definition for IEnumerable<T> – cannot have
access modifiers

  IEnumerator<GoObject> IEnumerable<GoObject>.GetEnumerator() {

    return this.GetEnumerator();  // need "this." to disambiguate

  }

  // this is the definition seen by a use like:  foreach (GoObject obj
in aGoCollection) { . . . }

  public virtual GoCollectionEnumerator GetEnumerator() {

    . . .

  }

  . . .

}
```

IEnumerator and IEnumerator<T>

Here's how **IEnumerator<T>** compares with **IEnumerator**. The minor feature is that it adds inheriting from **IDisposable**.

IEnumerator	IEnumerator<T> : IDisposable, IEnumerator
	IDisposable.Dispose()
MoveNext()	IEnumerator.MoveNext()
Reset()	IEnumerator.Reset()

Current	IEnumerator.Current *of type Object*
	Current *of type T*

ICollection and ICollection<T>

ICollection<T> is quite different from **ICollection**. The non-generic **ICollection** only provided read-only access to a collection. The generic **ICollection<T>** includes methods such as **Add** and **Clear**. The generic **ICollection<T>** also gets rid of **IsSynchronized** and **SyncRoot**, members that had always seemed superfluous.

The generic **ICollection<T>** also adds the **IsReadOnly** property. This was an issue for those sample **GoDocument**-derived classes that defined their own **IsReadOnly** property. As an expediency, we have just changed those definitions to "override" the one now provided by **GoDocument**.

For **IGoCollection** we have removed the members **CopyTo(Array, int)**, **IsSynchronized**, and **SyncRoot**, despite the additional incompatibility that that causes. We removed **CopyTo(Array, int)** to reduce type errors and to improve efficiency. And nobody cares about synchronization.

Since **IGoCollection** now implements **IEnumerable<GoObject>**, you can now use LINQ on **GoDiagram** collections, if you are using .NET 3.5. (But we can't in our libraries, since they need to compile/run with .NET 2.0.)

ICollection : IEnumerable	ICollection<T> : IEnumerable<T>, IEnumerable	IGoCollection : ICollection*<T> now inherits from ICollection<GoObject>*
	Add(T)	Add(GO)
	Clear()	Clear()
	Contains(T)	Contains(GO)
		CopyArray()
CopyTo(Array, int)	CopyTo(T[], int)	CopyTo(GO[], int) *removed Array overload*
IEnumerable.GetEnumerator()	IEnumerable.GetEnumerator()	GetEnumerator() *returns IEnumerable*
	IEnumerable<T>.GetEnumerator()	GetEnumerator() *added, returns IEnumerator<GO>*
		GetEnumerator() *returns GoCollectionEnumerator*
	Remove(T)	Remove(GO)

		Backwards
Count	Count	Count
		IsEmpty
	IsReadOnly	IsReadOnly *added*
IsSynchronized		*removed*
SyncRoot		*removed*

IList and IList<T>

IList<T> is reasonably similar to **IList**. The main annoyance in the conversion is that we had to change all of the overrides of **Remove** to return a boolean in all of the sample classes. **ArrayList** was the principal implementation of **IList**. We have already replaced practically all uses of **ArrayList** with the appropriate **List<T>.**

IList : ICollection, IEnumerable	IList<T> : ICollection<T>, IEnumerable<T>, IEnumerable
Add(O)	ICollection.Add(T)
Clear()	ICollection.Clear()
Contains(O)	ICollection.Contains(T)
ICollection.CopyTo(Array, int)	ICollection.CopyTo(T[], int)
IEnumerable.GetEnumerator()	IEnumerable.GetEnumerator() *returns IEnumerator*
	IEnumerable<T>.GetEnumerator() *returns IEnumerator<T>*
IndexOf(O)	IndexOf(T)
Insert(int, O)	Insert(int, T)
Remove(O) *returns void*	ICollection.Remove(T) ***returns bool***
RemoveAt(int)	RemoveAt(int)
ICollection.Count	ICollection.Count
IsFixedSize	
IsReadOnly	ICollection.IsReadOnly
ICollection.IsSynchronized	
Item[int] *indexer of type O*	Item[int] *indexer of type T*
ICollection.SyncRoot	

IDictionary and IDictionary<K,V>

IDictionary<K,V> is only moderately similar to **IDictionary**. Dictionaries use the generic **KeyValuePair<K,V>**, instead of **DictionaryEntry**. Enumeration

uses **IEnumerator<KeyValuePair<K,V>>**, instead of **IDictionaryEnumerator.**
Hashtable was the principal implementation of **IDictionary**. We have just replaced most
uses with the appropriate **IDictionary<K,V>**.

However, the biggest change is in how to look up values. Non-generic code is used to
doing:

```
Hashtable ht = …
MyClass c1 = (MyClass)ht["one"];
// now c1 is either null or has a MyClass object reference
```

But this won't work correctly any more. The main reason is that if there is no key
"one" in the hashtable, it will raise an exception. You could do:

```
Dictionary<String, MyClass> ht = …
MyClass c1 = null;
if (ht.Contains("one")) c1 = ht["one"];
```

But this is relatively inefficient, since if the key is present it requires two lookups.
Instead do:

```
Dictionary<String, MyClass> ht = …
MyClass c1;
ht.TryGetValue("one", out c1);
```

TryGetValue return true if it finds the key, false if it does not. If it finds the key,
the out parameter is set to the value it finds. (Note no need for the cast.)

If the key is not present, the out parameter is set to the default value for the value
type. Since MyClass is a reference type, it would be set to null.

IDictionary : ICollection, IEnumerable	IDictionary<K,V> : ICollection<KeyValuePair<K,V>>, IEnumerable<KeyValuePair<K,V>>, IEnumerable
	ICollection.Add(KeyValuePair<K,V>)
Add(O, O)	Add(K, V)
Clear()	ICollection.Clear()
Contains(O)	ContainsKey(K)
	ICollection.Contains(KeyValuePair<K,V>)
ICollection.CopyTo(Array, int)	ICollection.CopyTo(KeyValuePair<K,V>[], int)
IEnumerable.GetEnumerator()	IEnumerable.GetEnumerator()
GetEnumerator()	IEnumerable<KeyValuePair<K,V>>.GetEnumerator()

returns IDictionaryEnumerator	_returns IEnumerator<KeyValuePair<K,V>>_
	ICollection.Remove(KeyValuePair<K,V>)
Remove(O)	Remove(K)
	TryGetValue(K, out V) **_use this when both testing presence of key and getting the value_**
ICollection.Count	ICollection.Count
IsFixedSize	
IsReadOnly	ICollection.IsReadOnly
Item[O]	Item[K] **_throws exception when not present!_**
Keys	Keys
Values	Values

General Comments

There are still a number of uses of **System.Collections** in the **GoDiagram** libraries. Because System.**Collections.Generic.IEnumerable<T>** inherits from **System.Collections.IEnumerable**, and because **System.Collections.Generic.IEnumerator<T>** inherits from **System.Collections.IEnumerator**, there necessarily are some uses of **System.Collections** types, particularly with explicit interface definitions.

But even ignoring those cases, there remain some uses of **System.Collections**.

GoCopyDictionary still inherits from **Hashtable**. Because a lot of code, particularly in overrides of **GoGroup.CopyChildren**, makes use of the indexer lookup of newly copied objects given an original object, We didn't want to require everyone to reimplement that code to make use of **TryGetValue**.

Similarly, in **GoDiagram Web**, the table of parsed arguments that is passed to **GoViewDataRenderer.HandleClientRequest** is still a **Hashtable** instead upgrading to use a **Dictionary<String, String>**. Again, the same compatibility reasoning applies.

GoText.Choices used to be of type **ArrayList**. That was a design mistake. We should not have made the implementation type of the list a part of the API. It should have been of type **IList**. This permits a much broader range of data types that programmers can use to initialize the list of choices for a combobox.

But for that reason, we should leave the type of the property to be **IList**. Changing the type to be generic, presumably to **IList<Object>** or worse to **IList<String>**, would be counterproductive since that would unnecessarily constrain the kinds of data that programmers could use.

The same reasoning applies to other cases where the data is should be of type **Object** – to use **IEnumerable** or **IList.**
Examples: **Northwoods.Go.Xml.XmlWriter.Objects**, **Northwoods.Go.Xml.XmlReader**

.RootObject which might be of type **IList**,
and**Northwoods.Go.Instruments.GraduatedScale.LabelChoices**.

Upgrading Hints

If you are upgrading your application to use **GoDiagram** version 4 from version 3, you should find the task to be fairly easy.

First, add the following statement to your source files that use collections:

```
using System.Collections.Generic;
```

Second, consider these important .NET collection differences when upgrading your application:

- **ICollection<T>** includes collection-modifying methods that **ICollection** does not have

- **ICollection<T>.Remove** now returns a boolean – you may have some overrides of **GoGroup.Remove** that you'll need to update

- **IDictionary<K,V>.Item** gets now throw an exception if the key is not found – use **TryGetValue** instead