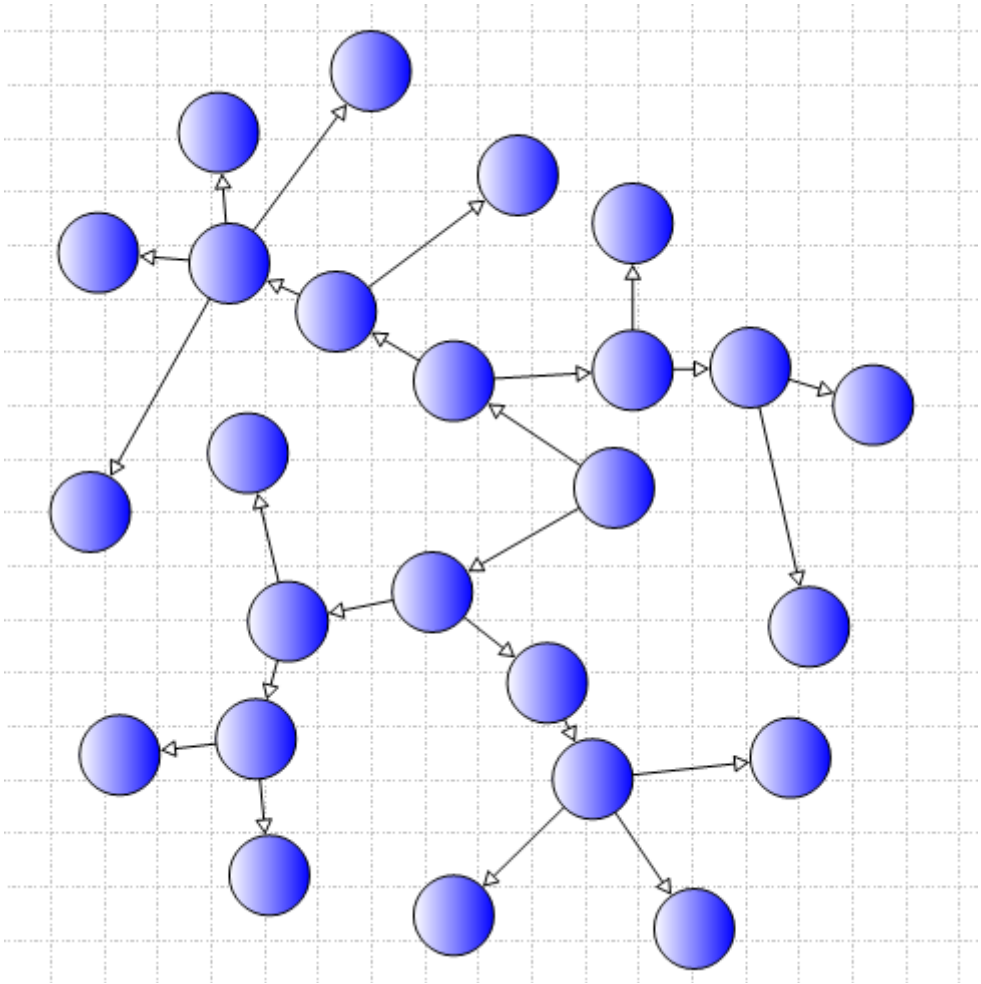
AddFlow for WinForms 2015 Tutorial



May 2015

Lassalle Technologies

http://www.lassalle.com

Contents

1 Introduction

AddFlow for WinForms 2015 is a general purpose Flowcharting/Diagramming .NET Windows Forms control, which lets you quickly build flowchart-enabled .NET applications.

AddFlow for WinForms 2015 allows the creation and the manipulation of two-dimensional diagrams (a.k.a graphs). An AddFlow diagram is a set of objects called nodes (also called vertices or entities) that can be linked each other with links (also called edges, arcs or relations). These diagrams can be created programmatically or interactively.

Each time you need to graphically display interactive diagrams, you should consider using AddFlow, a royalty-free control that offers unique support to create diagrams interactively or programmatically: workflow diagrams, database diagrams, communication networks, organizational charts, process flows, state transitions diagrams, CTI applications, CRM (Customer Relationship Management), expert systems, graph theory, quality control diagrams, …

The benefits of using AddFlow for WinForms 2015 are the following:

- Small deployment assembly. The size of the Lassalle.Flow.dll file is less than 250 Kb.
- Small programming interface: we have always preferred the quality to the quantity. We refuse to provide an inflation of classes and properties.
- Full integration with the .NET environment.
- Great Flexibility.
- Runtime royalty free.
- If you own a license of AddFlow for WinForms 2015, you have the option to purchase also a source code license.

Purpose of this tutorial

This tutorial provides informationon:

- creating diagrams programmatically, using the AddFlow control and classes
- creating diagrams interactively
- installing AddFlow for WinForms 2015
- licensing

Who should use this tutorial?

This guide is intended for application programmers using the .NET platform to build Windows Forms applications.

Samples

AddFlow for WinForms 2015 is installed with one demo sample written in C#: **demo.exe**. Its source code is installed too.

2 What's new in AddFlow for Winforms 2015 ?

AddFlow For Winforms 2015 is **NOT compatible** with previous versions v1 and v2.

What has changed ? Almost everything.

Why ? Those changes were necessary:

- to allow a better evolution of the product for the future
- to fix some design errors
- to provide a lighter product that just does its job and nothing more.
- to take account of some new technologies like LINQ.
- for a better compatibility with other versions, especially the HTML5 version of AddFlow.

Does it provide the same features as previous versions ? Some features have been removed.

Does it provide new features ? Yes of course:

- Pins. A Pin allows creating a link from the pin of a node to the pin of another node.
- Captions. A Caption is like a label or a note. It is a new type of object that allows displaying a text or an image and that can be owned by any item.
- Orthognal links . A new link line style.
- Quadtree. If the IsQuadTree property is set, the AddFlow items are managed in a Quadtree data structure that allows finding then quickly (useful only with very big diagrams)

Is it possible to load diagrams created with previous versions ? Yes but with some restrictions (children nodes)

A conversion guide can be found at the end of this tutorial.

3 Getting Started

3.1 Installation

The AddFlow for WinForms 2015 installation package is a Windows Installer file. It is the same file for the evaluation version and the full version. However, when you install it, you install the evaluation version. As explained in the Licensing section if you purchase the product, you will receive a license key allowing turning the evaluation version into the full version.

In the AddFlow for WinForms 2015 installation folder, it creates 3 subdirectories: Bin, Doc and Demo:

- The **Bin** subdirectory contains the assemblies: dlls, demo executable.
- The **Doc** subdirectory contains the help file, the tutorial, the readme file and the license agreement.
- The **Demo** subdirectory contains the C# source code of the Demo sample that demonstrates AddFlow for WinForms 2015.

3.2 AddFlow dlls

There are two dlls.

Assembly	**Description**
Lassalle.Flow.dll	The AddFlow control
Lassalle.Flow.Layout.dll	The LayoutFlow dll is a set of graph layout algorithms. This extension is not free. You must purchase a Professional license of AddFlow for WinForms 2015 to be able to use it without any restriction.

TIP: Is the source code available?

The source code of AddFlow and LayoutFlow is not provided. However you may purchase a source code license. A source code license agreement is installed with AddFlow.

Notice that all these assemblies are written in C#.

3.3 Licensing

3.3.1 Key Points

The key points are the following:

- each product is licensed **per individual developer**
- each product is **runtime royalty free**
- the evaluation version of each product has a **nag banner**. You may use it for up to **30 days** for trials and design-time evaluation purposes only.
- you may purchase either the **Standard Edition** of AddFlow, either the **Professional Edition** of AddFlow. The professional version provides also a set of graph layout algorithms.
- multi-pack discounts available
- It is also possible to purchase the **source code**.

3.3.2 Type of licenses

3.3.2.1 Editions

You may purchase either:

- an AddFlow for WinForms 2015 **Standard** Edition license. This license does not include LayoutFlow. If you try to execute a graph layout algorithm, you will face sometimes a nag screen.
- an AddFlow for WinForms 2015 **Professional** Edition license. This license includes LayoutFlow. You can execute the graph layout algorithms provided by LayoutFlow without any restriction.

3.3.2.2 Multi-pack discounts

We offer the following type of licenses:

- Single developer license: allows just one developer
- Team license: allows 4 developers
- Site license: allows unlimited developers at a single physical address.
- Enterprise license: allows all developers of an enterprise

3.3.2.3 Source code

You may also purchase the source code or not. A source code license agreement is installed with AddFlow. This agreement mainly says that you do not have the right to make a competitor product and don't have the right to divulge the code. You may purchase a source code license for AddFlow for Winforms Standard Edition or for AddFlow for Winforms Professional Edition. Please note that a source code license for AddFlow for Winforms Standard Edition requires the purchaser to own an AddFlow for Winforms Standard Edition license and that a source code license for AddFlow for Winforms Professional Edition requires the purchaser to own an AddFlow for Winforms Professional Edition license.

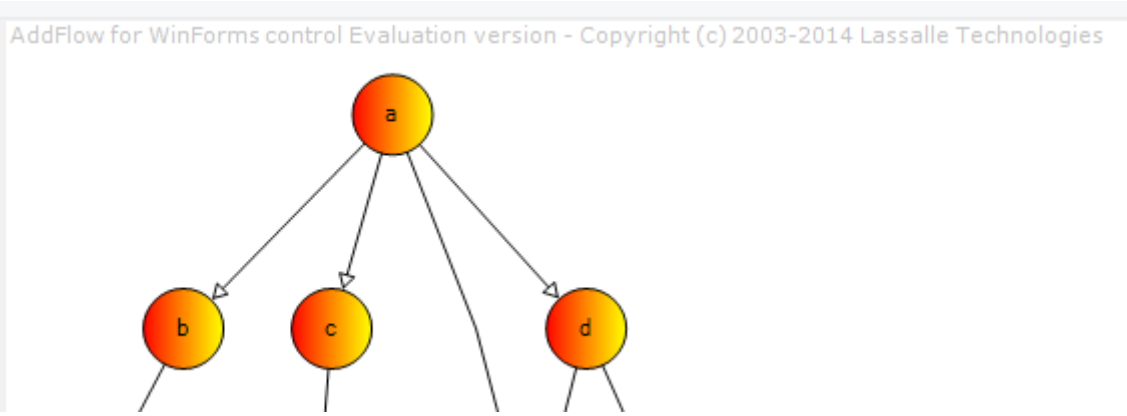
3.3.3 How it works?

The evaluation version

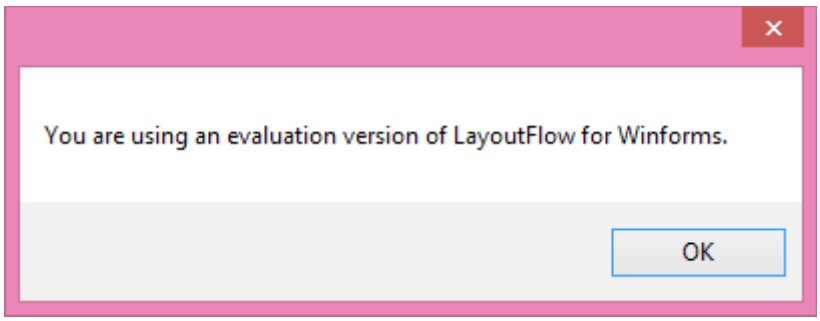
When you install AddFlow for WinForms 2015, you install in fact an evaluation version of AddFlow. (And you install also an evaluation version of LayoutFlow)

If you generate ("compile") an application that uses this evaluation version of AddFlow for WinForms 2015, then any attempt to use this application will display an evaluation label explaining that it has been generated only with an evaluation version of AddFlow.

In the following example, you can see the evaluation label displayed at the top of the diagram.



And if you execute one of the graph layout methods provided by the LayoutFlow dll, you will face sometimes a nag screen:



Remark: You may use the Evaluation Version of the Software for up to **30 days** for trials and design-time evaluation purposes only.

The full version

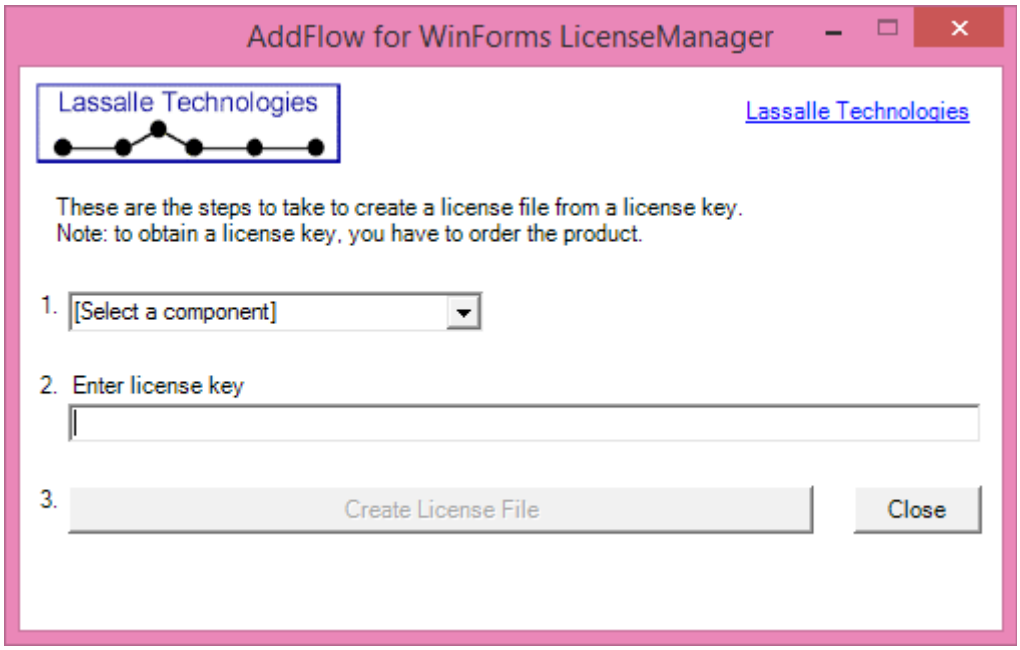
To get the full version of AddFlow, you have to purchase an AddFlow license. In such a case, you will receive an AddFlow license key (also called serial number or license number).

Now, it depends of the type of license you have purchased.

- If you have just purchased a Standard license of AddFlow for WinForms 2015, the license key will alllow you removing the evaluation label. However you will continue facing the nag screen if you try to execute a graph layout method.
- If you have just purchased a Professional license of AddFlow for WinForms 2015, no nag screen will be displayed if you execute a graph layout methods.

The LicenseManager program

The LicenseManager.exe program, provided with AddFlow for WinForms 2015, allows generating a license file from a license key.



The name of the license file is **Lassalle.Flow.AddFlow.LIC**. The license file is created in the same directory as the LicenseManager application, therefore in the bin subdirectory of the AddFlow for WinForms 2015 installation folder.

When developing your application, this license file Lassalle.Flow.AddFlow.LIC must be in the same directory as the licensed assembly Lassalle.Flow.dll.

Example

After having installed AddFlow for WinForms 2015, load the **Demo** solution (**demo.sln**) provided with AddFlow, then compile it and run it: the evaluation label will appear.

Now place the AddFlow license file in the bin subdirectory of the AddFlow for WinForms 2015 installation folder (This is possible only if you have already created a license file with the LicenseManager program, therefore if you have already a license key, therefore if you have purchased the product!). Then re-compile the application (Rebuild All) and run it again: there is not any evaluation label this time. And if it is a Professional license of AddFlow for WinForms 2015, no nag screen will be displayed when you attempt to execute a graph layout method.

Licenses.licx file

Notice the file licenses.licx in the **Demo** project. This file is a text file that identifies which licensed classes are used in a project. There should be one licenses.licx file (as an embedded resource) for each project in a VS.NET solution. Without this file, the licensing would not work. It must contain the assembly qualified name of AddFlow.

The line in the licenses.licx file that corresponds to AddFlow is the following:

```
Lassalle.Flow.AddFlow, Lassalle.Flow, Version=3.0.0.0, Culture=neutral,
PublicKeyToken=bfc5c756e54a9d2a
```

If you use the toolbox to place an AddFlow control on your application form, the file licenses.licx is automatically created. Otherwise, you can create it manually.

3.3.4 Licensing problems

If, after having obtained a license for AddFlow and having followed the instructions, you have still a nag screen, don't worry: you are not the first! Despite the fact that the licensing technology used in AddFlow is 100% standard and that it follows 100% the .NET guidelines, many customers are facing such a problem. It was also my case at the beginning. The following remarks may help you solving this licensing issue.

1) First of all, be sure that your project contains a licenses.licx file (as an embedded resource) as described before.

2) License keys are only recognized when they are linked into an executable assembly (.EXE), not a library (.DLL). Therefore, if you use AddFlow in another DLL (for instance a control library), then you will have to:

- include a reference to AddFlow in the application that uses that DLL (even though this executable did not directly use the AddFlow control)
- be sure that the licenses.licx file of the application itself contains a line referencing AddFlow.

4 Interactive creation of a diagram

4.1 Overview

The interactive creation of diagrams is mouse-based. It includes:

- the creation of nodes and links (including reflexive links)
- the selection of nodes and links (including multi-selection)
- the resizing of nodes
- the moving of nodes
- the stretching of links (the possibility to add or remove segments in a link)
- the possibility to change the origin or the destination of a link

It supports also scrolling and grids.

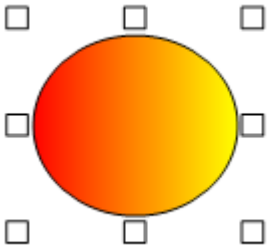
Moreover, many properties allow customizing the interactive behavior of an AddFlow control. For instance, you can prevent the user to create reflexive links with the **CanReflexLink** property or to move nodes with the **CanMoveItems** properties.

And a set of methods and properties allow implementing a powerful Undo/Redo feature.

4.2 Create a diagram interactively

4.2.1 Draw a node

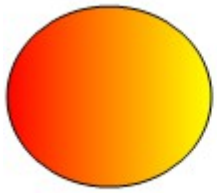
Bring the mouse cursor into the control, press the left button, move the mouse and release the left button. You have created an elliptic node. This node is selected: that's why 8 handles (little squares) are displayed.

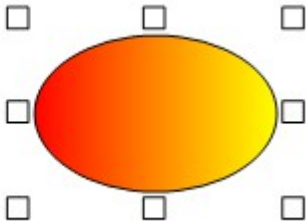


The 8 handles allow **resizing the node**. If you want to **move the node,** you bring the mouse cursor into the node, press the left button, move the mouse and release the left button. Notice that AddFlow allows you changing the size of the selection handles.

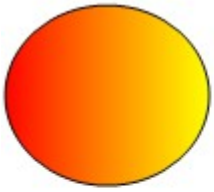
4.2.2 Draw a link

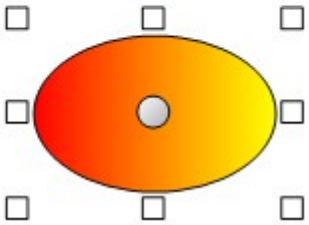
Draw a second node.





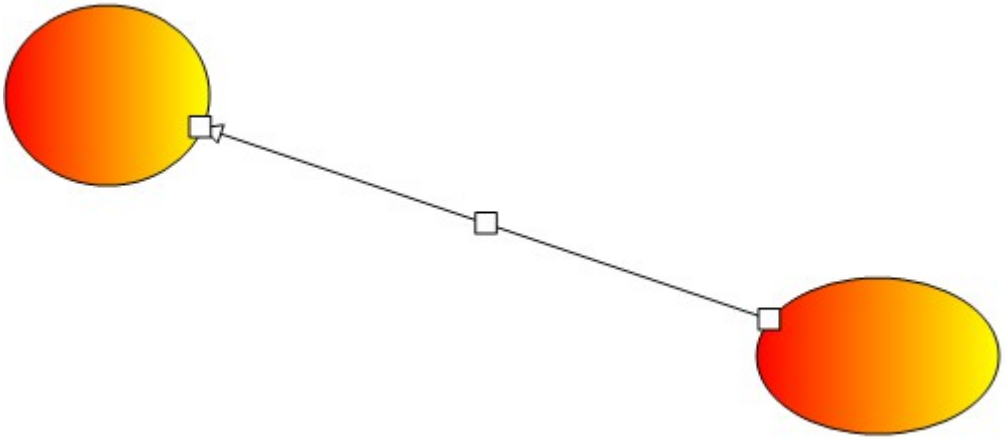
Then bring the mouse cursor above the second node. A small circle handle is then displayed at the center of the





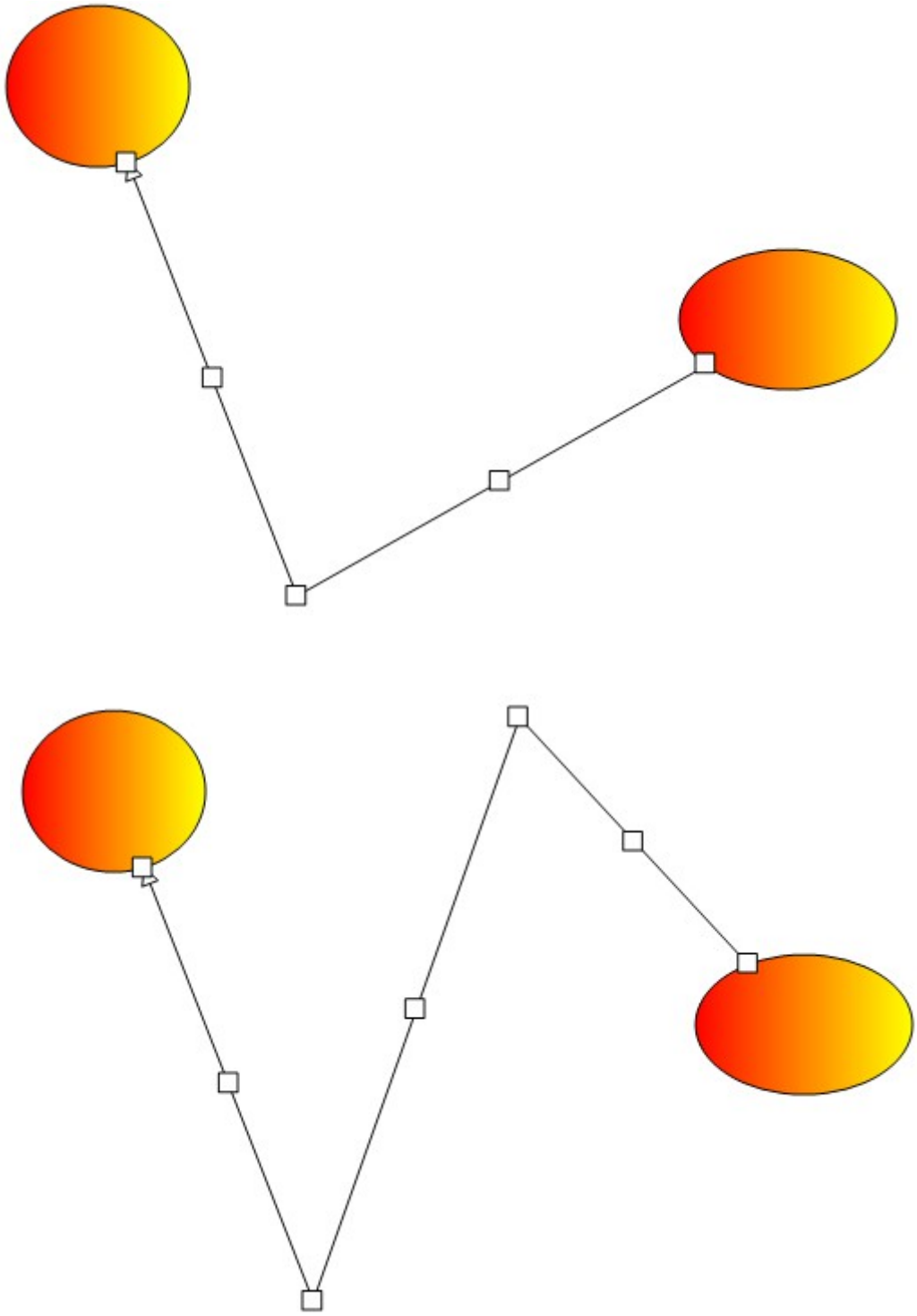
selected node.

Bring the mouse over this small circle handle, press the left button, move the mouse towards the other node. When the mouse cursor is into the other node, release the left button. The link has been created. And it is selected: 3 handles are displayed in the link.



4.2.3 Stretch a link

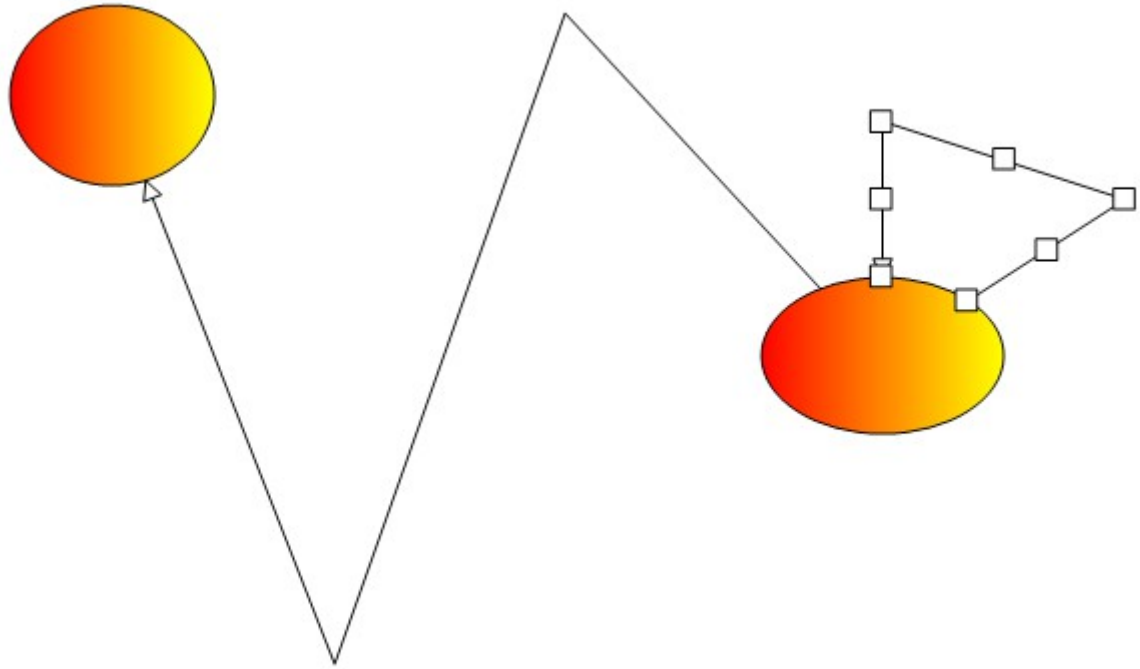
Bring the mouse cursor into the link handle in the middle of the link, press the left button, move the mouse and release the left button. You have created a new link segment. It has now 5 handles allowing you to add or remove segments. (The handle at the intersection of two segments allows you to remove a segment: you move it with the mouse so that the two segments are aligned and when these two segments are approximately aligned, release the left button).



Create another segment

4.2.4 Draw a reflexive link

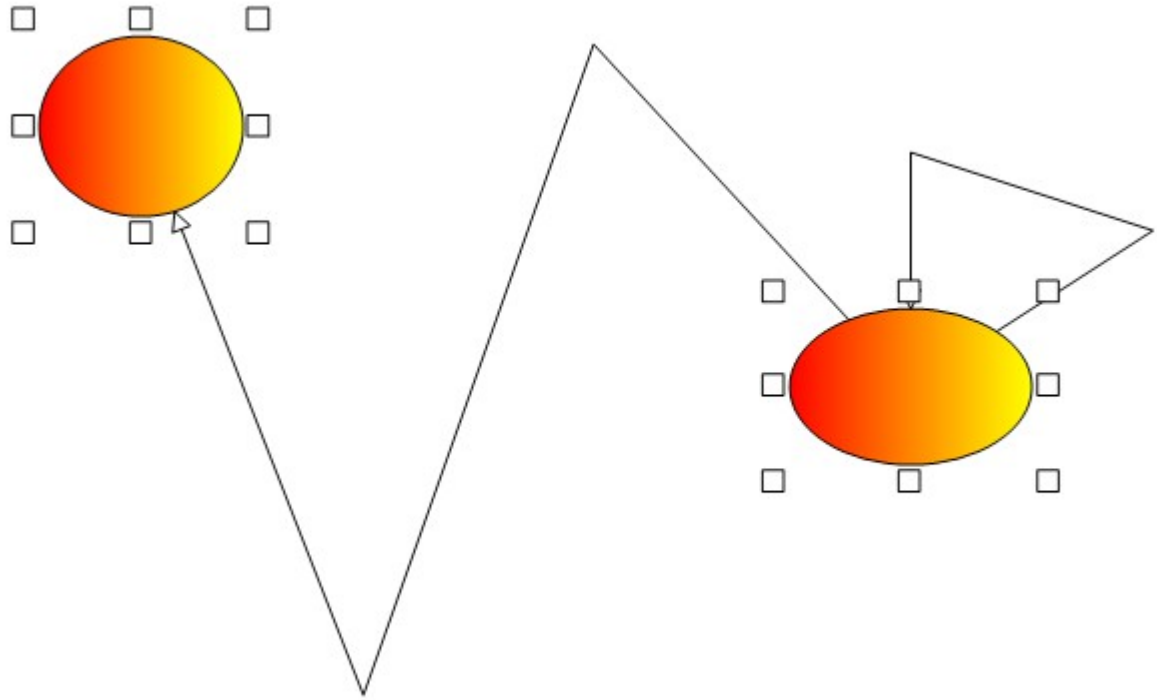
Select a node by clicking on it. Then bring the mouse cursor inside the selected node. Press the left button, move the mouse outside the selected node, then move it inside the selected node again, then release the left



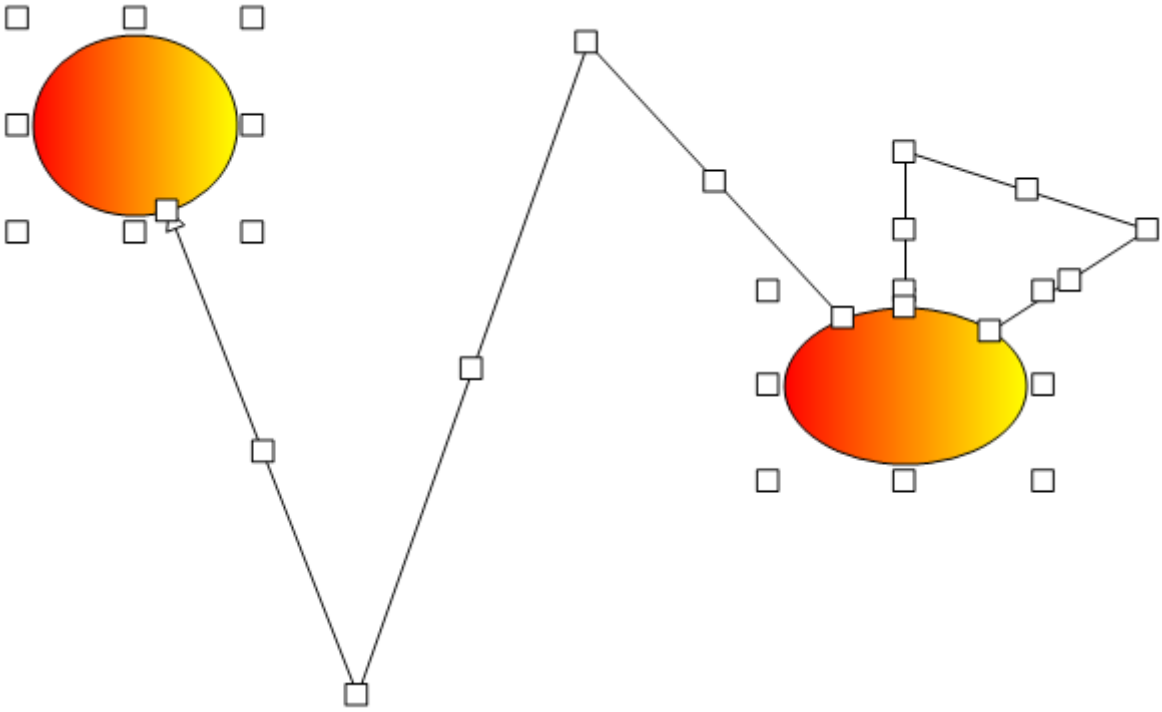
button. You have created a reflexive link, i.e. a link whose origin and destination are the same.

4.2.5 Multiselection

You can select several nodes by clicking them with the mouse and simultaneously pressing the shift or control key.



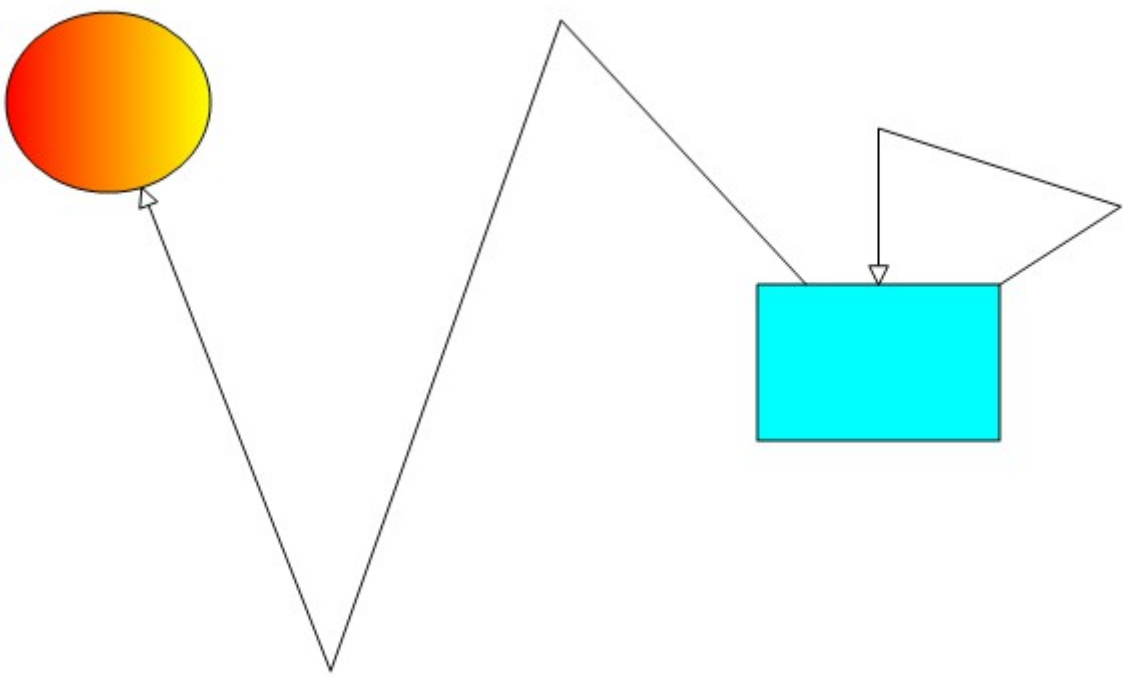
You can also select links or nodes and links.



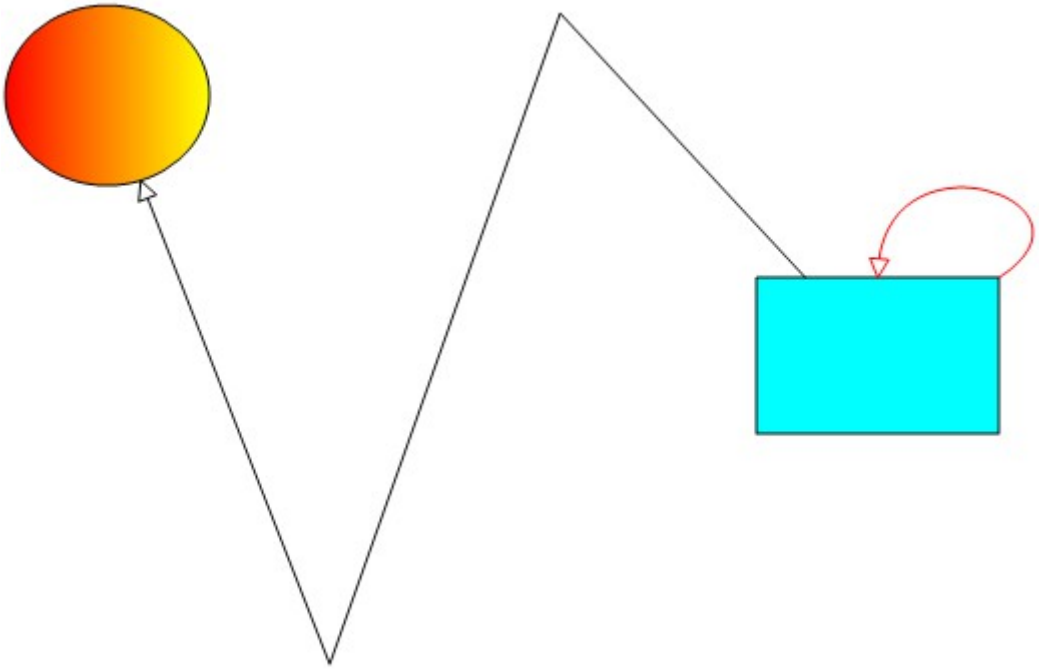
There is another way to perform multiselection, using the **MouseSelection** property and assigning it the **MouseSelection.Selection** value. Then you can select several nodes and links: you bring the mouse cursor into the AddFlow control, press the left button, move the mouse and release the left button. All nodes or links inside the selection rectangle are selected. Then you can unselect some nodes by clicking them with the mouse and simultaneously pressing the shift or control key. You can select them again by using the same method.

4.2.6 Change properties of a node or a link

Interactively, without adding any code, you can change the position and the size of a node (and also its text as described later). You can add segments to a link or remove them. To change the other properties (shape, styles, colors, behaviors, etc) of a node or a link, you have to write some code. For instance, you can add code that updates a property page when the user selects a node or a link. This is what is done in the **Demo** sample. If you click on the second node, its property page will appear and you can change some of its properties, for instance its shape and its filling color:

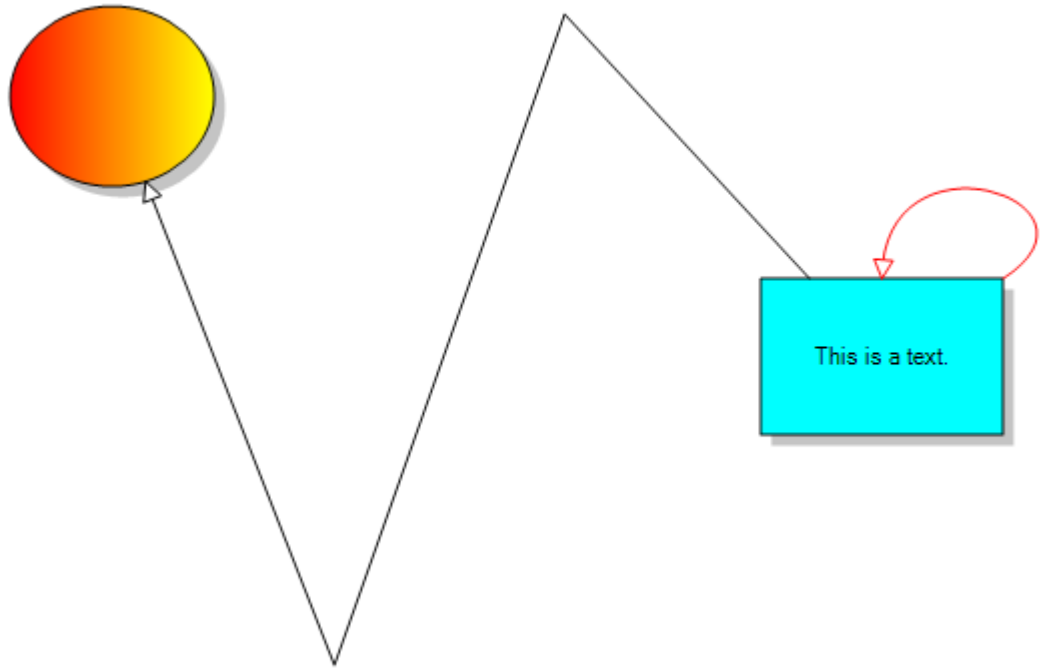


You can proceed the same way with, for instance, the reflexive link and change its style, its color:



4.2.7 Add a text to a node

Click for instance on the blue node. This will select it. Click another time. An edit box is displayed inside the node, allowing you to enter a text.

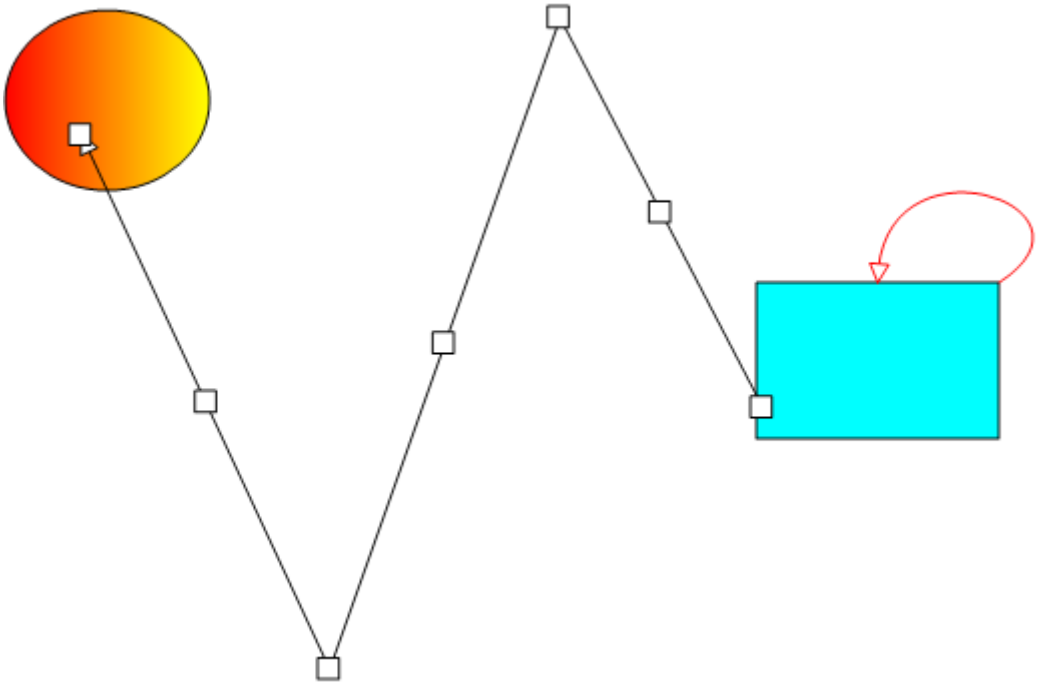


4.2.8 Adjust the link origin and destination points

In the previous example, you cannot adjust the extremities of the links. For instance, if you select the blue link and bring the mouse cursor into the last (or the first) handle of the blue link, press the left button, move the mouse in another place then relinquish the mouse button, the link springs back again, retrieving its initial position.
However, you two methods to change this behavior:

- by setting the **IsAdjustDst** and **IsAdjustOrg** properties to true.
- By using pins.

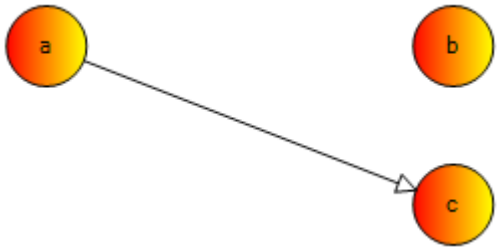
If you set the **IsAdjustDst** and **IsAdjustOrg** properties to true, then if you bring the mouse cursor, for instance, into the last handle of the link, press the left button, move the mouse and release it, you'll see that you have defined a new destination position for the link. If you move the destination node, the new link destination position keeps on following the node.



4.2.9 Change the destination or the origin node of a link

You can change interactively the destination or the origin of a link. For instance, consider the following diagram.

You can adjust the last point of the red link. Move it from node b to node c: bring the mouse cursor into the third link handle (near the arrow head), press the left button, move the mouse until the node c and release the left button.
Then, the new destination of the link will be the node c (If you move the node c, the link will follow it).



5 Programmatic creation of a diagram

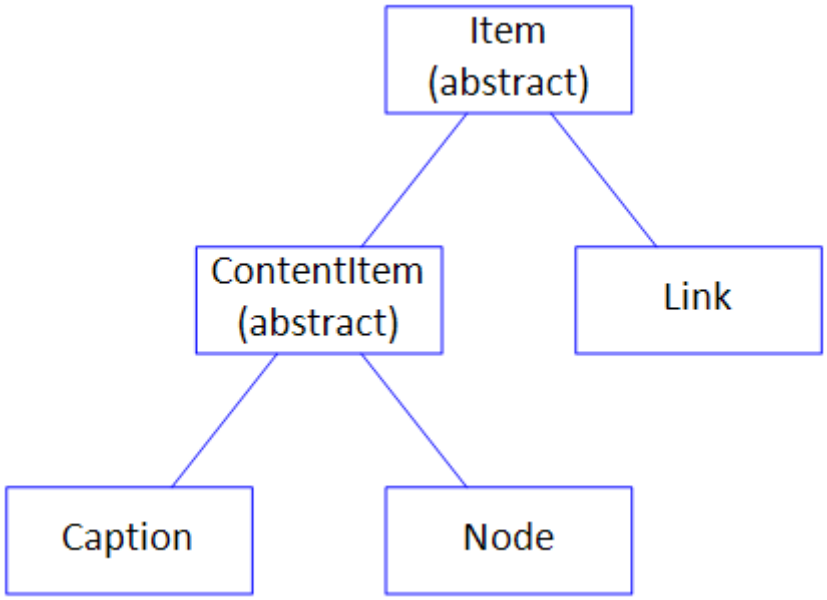
5.1 Overview

In this chapter we will focus on how to create a diagram programmatically.

The AddFlow library is a .NET class library containing a set of classes for creating interactive diagrams very easily.

The main class is the **AddFlow** class that derives from the .NET UserControl class. An AddFlow diagram contains three kinds of objects, **Node, Link** and **Caption** objects. The Node and Caption classes derives from the **ContentItem** class. The ContentItem and Link classes derive from the **Item** class.

The Item and ContentItem classes are abstract classes. You cannot use it directly but instead use objects that derive from the Node, Link and Caption classes.



For detailed information about the types, classes and interfaces used in AddFlow, see the **Lassalle.Flow** namespace in the AddFlow help file.

5.2 AddFlow Items

5.2.1 Item

The Item class represents an item in the diagram. All classes representing diagram elements derive from Item.

The main purpose is to provide common methods and properties of every diagram element: nodes, links, captions.

Note that it is an abstract class. You cannot use it directly but instead use objects that derive from the Node, Link or Caption classes.

The Item class provides some style properties, data properties and behavior properties used by all the items of an AddFlow diagram.

Style properties

- The **ShadowStyle, ShadowColor, ShadowSize** properties allow defining a shadow.
- The **FillColor**, **DrawColor**, **TextColor**, **GradientColor** properties allow defining item colors.

- The **Font** property allows defining the font used to display the text associated to the item.
- The **DashStyle**, **Thickness, IsOwnerDraw, Gradient** properties allows defining how the item is displayed.

Data properties

- The **Text** property defines the string displayed inside or near the item.
- The **Tooltip** property defines the tooltip of the item.
- The **Tag** property allows associating an object to an item.

Behaviour properties

IsSelected, **IsHitTestVisible, IsSelectable**, **ZOrder**, **IsInView.**

5.2.2 ContentItem

A ContentItem object is an Item object that has a content which may be a string or/and an image. Nodes and captions are ContentItem objects.

The main purpose is to provide common methods and properties for nodes and captions. Note that it is an abstract class. You cannot use it directly but instead use objects that derive from Node or Caption classes.

The **Location** and **Size** properties allow getting or setting the location and size of a ContentItem object.

The **ShapeFamily** and **GraphicsPath** propertyies allow getting ang setting the shape og the ContentItem object.

The **TextPosition**, **TextMargin, ImagePosition, ImageMargin** properties determine how the text and the image are positioned in the ContentItem object.

The **IsXMoveable**, **IsYMoveable**, **IsXSizeable**, **IsYSizeable** properties determine if the ContentItem object can be moved or resized.

The **IsEditable** properties determine if “in place” edition is possible in the ContentItem object.

5.2.3 Node

A node (also called vertice or entity) is a ContentItem object that can be linked to another node.

The Node class provides the **Links** collection property that allows getting all the links of the node.

The **IsInLinkable** and **IsOutLinkable** boolean properties determine if “in” or “out” links are allowed.

5.2.4 Link

A link (also called edge, relation or arc) is an Item object allowing linking two nodes. It is a line that leaves the origin node and comes to the destination node. A link cannot exist without its origin and destination nodes. If one of these two nodes is removed, the link is also removed.

The **Org** and **Dst** properties allow getting or setting the origin and destination node of the link.

The **PinOrgIndex** and **PinDstIndex** properties allow getting or setting the origin pin index and the destination pin index of the link.

The **Points** property is the collection of points that define the segments of the link.

The **LineStyle** property defines the style of link (polyline, Bezier, Spline, orthogonal)

The **IsArrowOrg**, **IsArrowDst** and **IsArrowMid** properties define if arrows are used for the link.

The **ArrowOrg**, **ArrowDst** and **ArrowMid** properties define the arrow shape.

The **JumpSize** property determines the size of the jump displayed at the intersection of 2 links.

The **RoundCornerSize** property determines the size of the rounded corners of the link segments.

The **IsOrientedText** property determines whether the link text can be drawn in the same direction as the link itself.

The **IsStretchable** property determines whether the link is stretchable or not. When a link is not stretchable, the user cannot interactively stretch it with the mouse.

The **IsAdjustDst** and **IsAdjustOrg** properties determineswhether it is possible to adjust the position of the last and first point of the link.

5.2.5 Caption

A caption (also called label or note) is a ContentItem object that can be owned by an AddFlow item, therefore by a node, a link or even by another caption.

The Caption class provides the **Owner** property that allows getting and setting the owner of the caption.

The **Dock** property returns/sets the DockStyle of the caption. This property is relevant only if the caption is attached to a ContentItem object.

The **AnchorPositionOnLink** property returns/sets a value which defines the position of the caption near the link. This property is relevant only if the caption is attached to a link.

5.3 Collections of items

AddFlow provide the following collections:

- **Items**: the collection of all items of the diagram
- **SelectedItems**: collection of all selected items of the diagram
- **Links**: collection of all links (in and out) of a node
- **Captions**: collection of captions of an AddFlow item.

WARNING: Those collections are provided for only for the AddFlow infrastructure and for enumeration purposes. Don't use them for adding or removing items.

5.4 Creation and deletion of items

5.4.1 Node

To add a node to a diagram, you have first to instanciate it then use the **AddNode** method to add it to the diagram. To remove it, you have juts to call the **RemoveNode** method.

The Node class has 3 constructors.

Node constructors

```
Node()
Node(float left, float top, float width, float height, string text, AddFlow addflow)
Node(float left, float top, float width, float height, string text, Node node)
```

We will often use the second constructor that creates a node with an initial position, an initial size, an initial text displayed inside the node and a reference to the AddFlow control that will supply default property values for the node.

The third constructor is working as the second one except that the default property values are copied from a node model.

5.4.2 Link

To add a link to a diagram, you have first to instanciate it then use the **AddLink** method to add it to the diagram. To remove it, you have juts to call the **RemoveLink** method.

The Link class has 5 constructors.

Link constructors

```
Link()
Link(Node org, Node dst, string text, Link link)
Link(Node org, Node dst, int pinOrgIndex, int pinDstIndex, string text, Link link)
Link(Node org, Node dst, string text, AddFlow addflow)
Link(Node org, Node dst, int pinOrgIndex, int pinDstIndex, string text, AddFlow addflow)
```

As for nodes, to supply default property values for the link, you can use a reference to the AddFlow control or use a link model.

5.4.3 Caption

To add a caption to a diagram, you have first to instanciate it then use the **AddCaption** method to add it to the diagram. To remove it, you have juts to call the **RemoveCaption** method.

The Caption class has 3 constructors.

Caption constructors

```
Caption()
Caption(float left,float top,float width,float height,string text,Item owner, AddFlow addflow)
Caption(float left,float top,float width,float height,string text,Item owner, Caption caption)
```

We will often use the second constructor that creates a caption with an initial position, an initial size, an initial text displayed inside the caption, a reference to the owner item of this caption and a reference to the AddFlow control that will supply default property values for the node.

The third constructor is working as the second one except that the default property values are copied from a caption model.

5.5 Diagram creation

5.5.1 Our first diagram

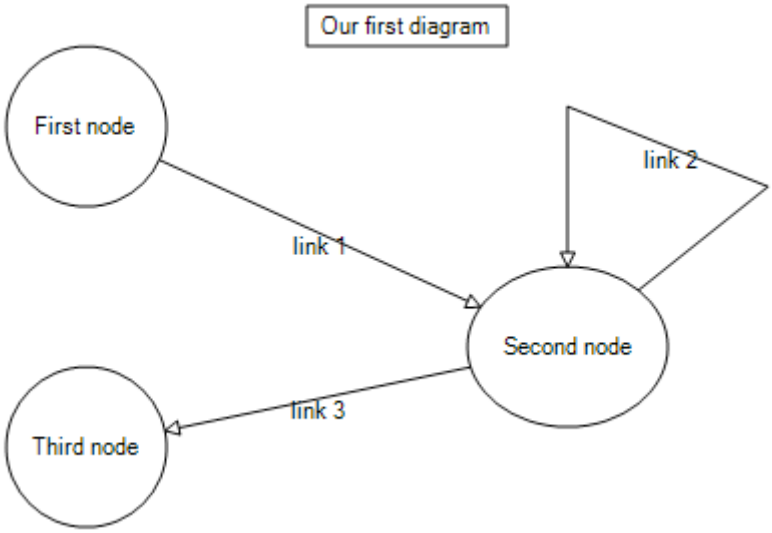
The **FirstPanel.cs** file of the **Demo** sample displays a small diagram that we will call “our first diagram”.

Following is the C# code that creates this simple diagram:

```
private void CreateDiagram(AddFlow addflow)
{
    addflow.Dock = DockStyle.Fill;
    addflow.AutoScroll = true;
    addflow.BackColor = SystemColors.Window;
    addflow.PageUnit = GraphicsUnit.Pixel;

    // Create 3 nodes
    Node node1 = new Node(50, 50, 80, 80, "First node", addflow);
    Node node2 = new Node(280, 160, 100, 80, "Second node", addflow);
    Node node3 = new Node(50, 210, 80, 80, "Third node", addflow);

    // Create 3 links
    Link link1 = new Link(node1, node2, "link 1", addflow);
    Link link2 = new Link(node2, node2, "link 2", addflow);
    Link link3 = new Link(node2, node3, "link 3", addflow);

    // Create 1 caption
    Caption caption = new Caption(200, 30, 100, 20, "Our first diagram", null,
addflow);

    // Add the items to the diagram
    addflow.AddNode(node1);
    addflow.AddNode(node2);
    addflow.AddNode(node3);
    addflow.AddLink(link1);
    addflow.AddLink(link2);
```

```
    addflow.AddLink(link3);
    addflow.AddCaption(caption);
}
```

This code creates the following diagram:



In this diagram, the nodes and links receive default property values. For instance, the nodes have an elliptical shape. The links are composed of one line terminated by an arrow. The link 2 is reflexive and by default, it is created with 3 segments. The drawing color is black. The text color is black.

We are going to enhance this diagram. However, let us focus on the way nodes, links and captions are created. First we create objects, then we add them to the AddFlow control. If we just write:

```
Node node1 = new Node(50, 50, 80, 80, "First node", addflow);
```

the node object is created. However it is not still part of the diagram. To make this node belong to the diagram, we have to add the following line:

```
addflow.AddNode(node1);
```

Same thing for links: if we just write:

```
Link link1 = new Link(node1, node2, "link 1", addflow);
```

the link object is created. However, to give a true existence to this link, we have to add the following line:

```
addflow.AddLink(link1);
```

5.5.2 Other ways to create the diagram

We could have written this first program in the following manner:

```
void CreateDiagram2(AddFlow addflow)
{
    addflow.Dock = DockStyle.Fill;
    addflow.AutoScroll = true;
    addflow.BackColor = SystemColors.Window;
    addflow.PageUnit = GraphicsUnit.Pixel;

    // Create and add the nodes to the diagram
    addflow.AddNode(new Node(50, 50, 80, 80, "First node", addflow));
    addflow.AddNode(new Node(280, 160, 100, 80, "Second node", addflow));
    addflow.AddNode(new Node(50, 210, 80, 80, "Third node", addflow));

    // We use LINQ to select in an array all the nodes of the canvas.
    var nodes = addflow.Items.OfType<Node>().ToArray();

    // Create and add the links to the diagram
```

```
    addflow.AddLink(new Link(nodes[0], nodes[1], "link 1", addflow));
    addflow.AddLink(new Link(nodes[1], nodes[1], "link 2", addflow));
    addflow.AddLink(new Link(nodes[1], nodes[2], "link 3", addflow));

    addflow.AddCaption(new Caption(200, 30, 100, 20, "Our first diagram", null,
addflow));
}
```

In this case, we use a **nodes** collection created with LINQ instead of using a reference for each Node object.

We could also use helpers functions to create nodes and links, as in the following CreateDiagram3 function where we use the helpers functions AddNode and AddLink.

```
void CreateDiagram3(AddFlow addflow)
{
    addflow.Dock = DockStyle.Fill;
    addflow.AutoScroll = true;
    addflow.BackColor = SystemColors.Window;
    addflow.PageUnit = GraphicsUnit.Pixel;

    // Create and add the nodes to the diagram
    Node node1 = this.AddNode(addflow, 50, 50, 80, 80, "First node");
    Node node2 = this.AddNode(addflow, 280, 160, 100, 80, "Second node");
    Node node3 = this.AddNode(addflow, 50, 210, 80, 80, "Third node");

    // Create and add the links to the diagram
    this.AddLink(addflow, node1, node2, "link 1");
    this.AddLink(addflow, node2, node2, "link 2");
    this.AddLink(addflow, node2, node3, "link 3");

    // Create and a caption to the diagram
    this.AddCaption(addflow, 200, 30, 100, 20, "Our first diagram", null);
}

Node AddNode(AddFlow addflow, float left, float top, float width, float height,
string text)
{
    Node node = new Node(left, top, width, height, text, addflow);
    addflow.AddNode(node);
    return node;
}

Link AddLink(AddFlow addflow, Node org, Node dst, string text)
{
    Link link = new Link(org, dst, text, addflow);
    addflow.AddLink(link);
    return link;
}

Caption AddCaption(AddFlow addflow, float left, float top, float width, float
height, string text, Item owner)
{
    Caption caption = new Caption(left, top, width, height, text, owner, addflow);
    addflow.AddCaption(caption);
    return caption;
}
```

In the **Demo** sample, we often use this kind of helpers functions.

Now we are going to enhance our diagram.

5.5.3 Changing property values

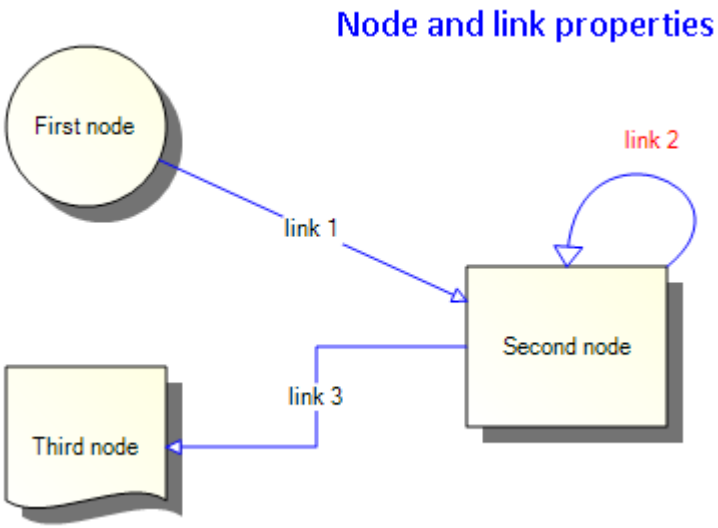
Now let us include the following “using” statement (it can be found in the **Demo** sample)

```
using Lassalle.Geometries;
```

and let us use the following diagram creation method (**PropertiesPanel.cs** file of the **Demo** sample):

```
private void CreateDiagram(AddFlow addflow)
{
    addflow.Dock = DockStyle.Fill;
    addflow.AutoScroll = true;
    addflow.BackColor = SystemColors.Window;
    addflow.PageUnit = GraphicsUnit.Pixel;

    // Create 3 yellow nodes with a shadow.
    // The second node is rectangular
    // and the third one has a Document shape style.
    Node node1 = new Node(50, 50, 80, 80, "First node", addflow);
    node1.FillColor = Color.LightYellow;
    node1.ShadowStyle = ShadowStyle.RightBottom;

    Node node2 = new Node(280, 160, 100, 80, "Second node", addflow);
    node2.FillColor = Color.LightYellow;
    node2.ShadowStyle = ShadowStyle.RightBottom;
    node2.ShapeFamily = ShapeFamily.Rectangle;

    Node node3 = new Node(50, 210, 80, 80, "Third node", addflow);
    node3.FillColor = Color.LightYellow;
    node3.ShadowStyle = ShadowStyle.RightBottom;
    node3.ShapeFamily = ShapeFamily.Rectangle;
    node3.GraphicsPath = PredefinedGeometry.GetNodePath(
                         NodeShapeStyle.Document, ShapeOrientation.so_0);

    // Create 3 links.
    // Each link is blue and its BackMode property set to Opaque.
    // The second link has a Bezier style, color of its text is red, and
    // its destination arrow head angle is 30°.
    // The third link has a "HVH" style.
    Link link1 = new Link(node1, node2, "link 1", addflow);
    link1.DrawColor = Color.Blue;
    link1.BackMode = BackMode.Opaque;

    Link link2 = new Link(node2, node2, "link 2", addflow);
    link2.DrawColor = Color.Blue;
    link2.BackMode = BackMode.Opaque;
    link2.LineStyle = LineStyle.Bezier;
    link2.TextColor = Color.Red;
    link2.ArrowDst = PredefinedGeometry.GetLinkArrowStyle(LinkArrowStyle.Arrow, 10,
14);

    Link link3 = new Link(node2, node3, "link 3", addflow);
    link3.DrawColor = Color.Blue;
    link3.BackMode = BackMode.Opaque;
    link3.LineStyle = LineStyle.Orthogonal;

    // Create 1 caption
    Caption caption = new Caption(200, 30, 220, 20, "Node and link properties",
null, addflow);
    caption.TextColor = Color.Blue;
    caption.DrawColor = Color.Transparent;
    caption.Font = new Font("Calibri", 14);

    // Add the nodes and the links to the diagram
    addflow.AddNode(node1);
    addflow.AddNode(node2);
    addflow.AddNode(node3);
    addflow.AddLink(link1);
    addflow.AddLink(link2);
    addflow.AddLink(link3);
    addflow.AddCaption(caption);
}
```

If we compile and execute this program, it will create the following diagram:



Now, our nodes have distinct shapes. They have a shadow. Their filling colour is LightYellow. And our links are blue. The reflexive link is a curved line, its text is red and the angle of its arrow head is larger. Even the caption has changed.

Therefore we know how to give distinct property values for each object.

Notice however that to specify the colour of each node, we had to do it for each node, even if the colour is the same. It is the same thing for the links. For a big diagram, this may be annoying to repeat always the same code for each object.

Fortunately, AddFlow allows using default property values that apply to all the next created nodes or links.

5.5.4 Default property values

Now, let us use the following diagram creation method (**DefaultPropertiesPanel.cs** file of the **Demo** sample):

```
private void CreateDiagram(AddFlow addflow)
{
    addflow.Dock = DockStyle.Fill;
    addflow.AutoScroll = true;
    addflow.BackColor = SystemColors.Window;
    addflow.PageUnit = GraphicsUnit.Pixel;

    // Default property values for nodes
    addflow.NodeModel.FillColor = Color.LightYellow;
    addflow.NodeModel.ShadowStyle = ShadowStyle.RightBottom;

    // Default property values for links
    addflow.LinkModel.DrawColor = Color.Blue;
    addflow.LinkModel.BackMode = BackMode.Opaque;

    // Default property values for captions
    addflow.CaptionModel.TextColor = Color.Blue;
    addflow.CaptionModel.DrawColor = Color.Transparent;
    addflow.CaptionModel.Font = new Font("Calibri", 14);

    // Create 3 yellow nodes with a shadow.
    // The second node is rectangular
    // and the third one has a Document shape style.
    Node node1 = new Node(50, 50, 80, 80, "First node", addflow);

    Node node2 = new Node(280, 160, 100, 80, "Second node", addflow);
    node2.ShapeFamily = ShapeFamily.Rectangle;

    Node node3 = new Node(50, 210, 80, 80, "Third node", addflow);
    node3.ShapeFamily = ShapeFamily.Rectangle;
```

```
    node3.GraphicsPath = PredefinedGeometry.GetNodePath(
                        NodeShapeStyle.Document, ShapeOrientation.so_0);

    // Create 3 links.
    // Each link is blue and its BackMode property set to Opaque.
    // The second link has a Bezier style, color of its text is red, and
    // its destination arrow head angle is 30°.
    // The third link has a "HVH" style.
    Link link1 = new Link(node1, node2, "link 1", addflow);

    Link link2 = new Link(node2, node2, "link 2", addflow);
    link2.LineStyle = LineStyle.Bezier;
    link2.TextColor = Color.Red;
    link2.ArrowDst = PredefinedGeometry.GetLinkArrowStyle(LinkArrowStyle.Arrow, 10,
14);

    Link link3 = new Link(node2, node3, "link 3", addflow);
    link3.LineStyle = LineStyle.Orthogonal;

    // Create 1 caption
    Caption caption = new Caption(200, 30, 220, 20, "Default properties", null,
addflow);

    // Add the items to the diagram
    addflow.AddNode(node1);
    addflow.AddNode(node2);
    addflow.AddNode(node3);
    addflow.AddLink(link1);
    addflow.AddLink(link2);
    addflow.AddLink(link3);
    addflow.AddCaption(caption);
}
```

If we compile and execute this new program, it will create the same diagram. However, our program is smaller because we have used the **NodeModel, CaptionModel** and the **LinkModel** properties of AddFlow which allow specifying default property values for nodes and links. For instance, writing:

```
addflow.NodeModel.FillColor = Color.LightYellow;
```

indicates that all the nodes that will be created after will be filled with a LightYellow color.

Then you just need to specify the property values that differ from the defaults.

Notice that the **NodeModel, CaptionModel and the LinkModel** properties have also an interactive effect. Not only the nodes created programmatically will be filled with a LightYellow colour but also the nodes created interactively with the mouse. This may be interesting or not, depending on what you intend to do.

Anyway, it is also possible to specify default values when creating a diagram programmatically and have other default values for the interactive creation.

5.5.5 The NodeModel, LinkModel and CaptionModel properties

We may also clone the NodeModel and the LinkModel properties respectively in a **Node** and a **Link** object and then we change some property values of these objects before using them when creating nodes and links, as in the following function:

```
private void CreateDiagram2(AddFlow addflow)
{
    addflow.Dock = DockStyle.Fill;
    addflow.AutoScroll = true;
    addflow.BackColor = SystemColors.Window;
    addflow.PageUnit = GraphicsUnit.Pixel;

    Node dn = new Node(0, 0, 0, 0, null, addflow);
    Link dl = new Link(null, null, null, addflow);
    Caption dc = new Caption(0, 0, 0, 0, null, null, addflow);

    // Default property values for nodes created programmatically
```

```
    dn.FillColor = Color.LightYellow;
    dn.ShadowStyle = ShadowStyle.RightBottom;

    // Default property values for links created programmatically
    dl.DrawColor = Color.Blue;
    dl.BackMode = BackMode.Opaque;

    // Default property values for captions created programmatically
    dc.TextColor = Color.Blue;
    dc.DrawColor = Color.Transparent;
    dc.Font = new Font("Calibri", 14);

    // Create 3 yellow nodes with a shadow.
    // The second node is rectangular
    // and the third one has a Document shape style.
    Node node1 = new Node(50, 50, 80, 80, "First node", dn);

    Node node2 = new Node(280, 160, 100, 80, "Second node", dn);
    node2.ShapeFamily = ShapeFamily.Rectangle;

    Node node3 = new Node(50, 210, 80, 80, "Third node", dn);
    node3.ShapeFamily = ShapeFamily.Rectangle;
    node3.GraphicsPath = PredefinedGeometry.GetNodePath(
                        NodeShapeStyle.Document, ShapeOrientation.so_0);

    // Create 3 links.
    // Each link is blue and its BackMode property set to Opaque.
    // The second link has a Bezier style, color of its text is red, and
    // its destination arrow head angle is 30°.
    // The third link has a "HVH" style.
    Link link1 = new Link(node1, node2, "link 1", dl);

    Link link2 = new Link(node2, node2, "link 2", dl);
    link2.LineStyle = LineStyle.Bezier;
    link2.TextColor = Color.Red;
    link2.ArrowDst = PredefinedGeometry.GetLinkArrowStyle(LinkArrowStyle.Arrow, 10,
14);

    Link link3 = new Link(node2, node3, "link 3", dl);
    link3.LineStyle = LineStyle.Orthogonal;

    // Create 1 caption
    Caption caption = new Caption(200, 30, 220, 20, "Default properties", null, dc);

    // Add the nodes and the links to the diagram
    addflow.AddNode(node1);
    addflow.AddNode(node2);
    addflow.AddNode(node3);
    addflow.AddLink(link1);
    addflow.AddLink(link2);
    addflow.AddLink(link3);
    addflow.AddCaption(caption);
}
```

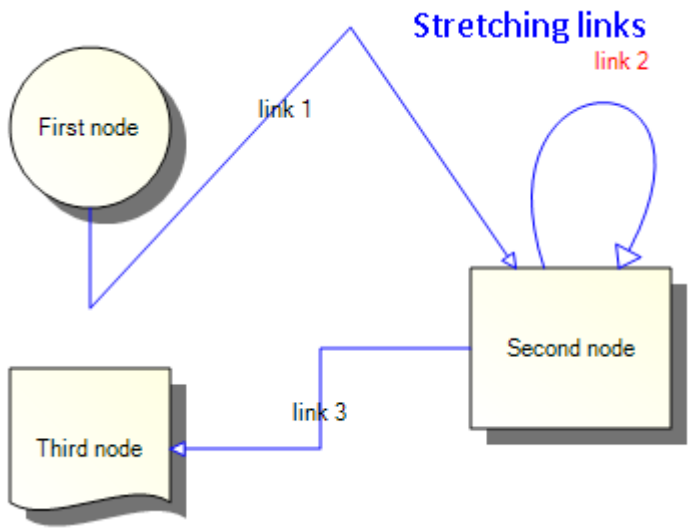
5.5.6 Stretching the links

We would like to add segments to our links. The following method (**StretchingLinksPanel.cs** file of the **Demo** sample) demonstrates how to do that.

```
private void CreateDiagram(AddFlow addflow)
{
    addflow.Dock = DockStyle.Fill;
    addflow.AutoScroll = true;
    addflow.BackColor = SystemColors.Window;
    addflow.PageUnit = GraphicsUnit.Pixel;

    Node dn = new Node(0, 0, 0, 0, null, addflow);
```

```
    Link dl = new Link(null, null, null, addflow);
    Caption dc = new Caption(0, 0, 0, 0, null, null, addflow);

    // Default property values for nodes created programmatically
    dn.FillColor = Color.LightYellow;
    dn.ShadowStyle = ShadowStyle.RightBottom;

    // Default property values for links created programmatically
    dl.DrawColor = Color.Blue;
    dl.BackMode = BackMode.Opaque;

    // Default property values for captions created programmatically
    dc.TextColor = Color.Blue;
    dc.DrawColor = Color.Transparent;
    dc.Font = new Font("Calibri", 14);

    // Create 3 yellow nodes with a shadow.
    // The second node is rectangular
    // and the third one has a Document shape style.
    Node node1 = new Node(50, 50, 80, 80, "First node", dn);

    Node node2 = new Node(280, 160, 100, 80, "Second node", dn);
    node2.ShapeFamily = ShapeFamily.Rectangle;

    Node node3 = new Node(50, 210, 80, 80, "Third node", dn);
    node3.ShapeFamily = ShapeFamily.Rectangle;
    node3.GraphicsPath = PredefinedGeometry.GetNodePath(
                         NodeShapeStyle.Document, ShapeOrientation.so_0);

    // Create 3 links.
    // Each link is blue and its BackMode property set to Opaque.
    // The second link has a Bezier style, color of its text is red, and
    // its destination arrow head angle is 30°.
    // The third link has a "HVH" style.
    Link link1 = new Link(node1, node2, "link 1", dl);

    Link link2 = new Link(node2, node2, "link 2", dl);
    link2.LineStyle = LineStyle.Bezier;
    link2.TextColor = Color.Red;
    link2.ArrowDst = PredefinedGeometry.GetLinkArrowStyle(LinkArrowStyle.Arrow, 10,
14);

    Link link3 = new Link(node2, node3, "link 3", dl);
    link3.LineStyle = LineStyle.Orthogonal;

    // Create 1 caption
    Caption caption = new Caption(200, 30, 220, 20, "Stretching links", null, dc);

    // Add the nodes and the links to the diagram
    addflow.AddNode(node1);
    addflow.AddNode(node2);
    addflow.AddNode(node3);
    addflow.AddLink(link1);
    addflow.AddLink(link2);
    addflow.AddLink(link3);
    addflow.AddCaption(caption);

    // Add 2 points (therefore 2 segments) to the first link
    link1.AddPoint(new PointF(90, 180));
    link1.AddPoint(new PointF(220, 40));

    // Stretch the reflexive link
    link2.SetPoint(new PointF(280, 50), 1);
    link2.SetPoint(new PointF(420, 50), 2);
}
```

If we compile and execute this program, it will create the following diagram:



The rules for managing the link collection of points are the following:

- The **Points** property of a link is its collection of points defining its segments. This property is only provided for the AddFlow infrastructure. so don't use it except for serialization purpose (as demonstraed in the file xmlflow.cs). To manipulate the collection of link points, you should use instead the methods **AddPoint**, **RemovePoints**, **ClearPoints**, **CountPoints**, **GetPoint** and **SetPoint**.
- You can manipulate the link points only after its insertion in the diagram.
- After its insertion in the diagram, a link has at least 2 points.
- You cannot remove these 2 points. The Count property of the Points collection is always superior or equal to 2. If you execute the ClearPoint method, then the CountPoint method returns 2.
- You can add or delete points only if the link line style is Polyline or Spline. In other case, the number of link points is fixed. For instance, if the link line style is Bezier, then it has 4 points in any case.
- You cannot change the first point of the Points collection except if the **IsAdjustOrg** property is true.
- You cannot change the last point of the Points collection except if the **IsAdjustDst** property is true.
- You can change each other point of the Points collection in any case.

5.6 More informations about ContentItem objects and nodes

A ContentItem object is an item that has a text and/or an image content. And its shape can be customized.

However, only nodes accept pins: we may define for each node a set of pins to connect links.

(Remember that nodes and captions are ContentItem objects.)

5.6.1 ContentItem colors

Five properties allow setting colors for a ContentItem object:

- **DrawColor** It is the color of the ContentItem object border.
- **FillColor** It is the ContentItem object filling color.
- **TextColor** It is the color of the ContentItem object text.
- **GradientColor** It is used in conjunction with the **FillColor** property to set a gradient color.
- **GradientMode** Returns/sets the direction of the linear gradient defined by the FillColor and GradientColor properties.

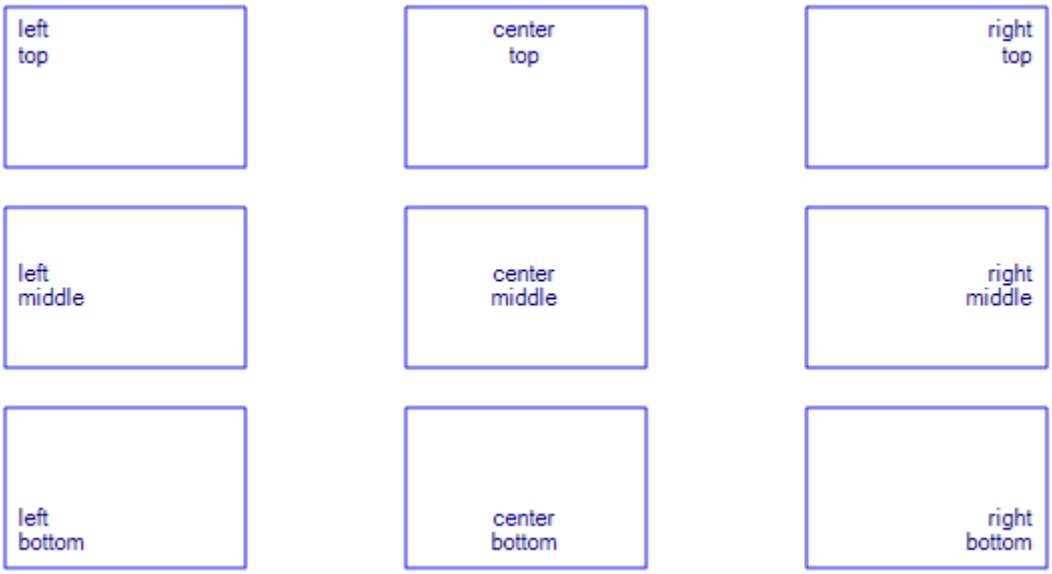
TIP: How to make the node's (or caption's) border transparent?

```
node.DrawColor = Color.Transparent;
```

5.6.2 ContentItem object text

You can associate a text to a ContentItem object with the **Text** property. Two properties allow placing the text inside the item: **TextPosition** and **TextMargin**. This is demonstrated in the **NodeTextPanel.cs** file of the **Demo** sample. In this example, the TextMargin is set to be equal to 5 at each side.

```
addflow.NodeModel.TextMargin = new System.Windows.Forms.Padding(5, 5, 5, 5);
```



The same could be done for a caption as it is also a ContentItem object.

5.6.3 ContentItem object custom shape

The **GraphicsPath** property allows you associating a custom shape to a ContentItem object. If this property is null, then the shape used is determined by the **ShapeFamily** property (ellipse or rectangle)

The following method (**customShapePanel.cs** file of the **Demo** sample) gives an example.

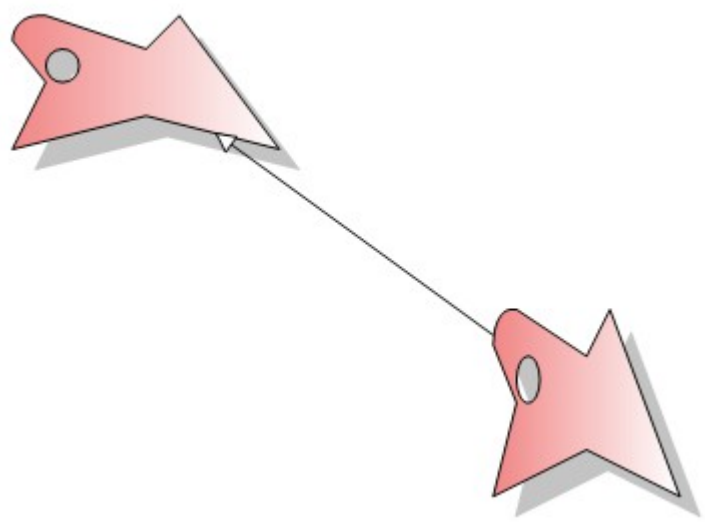
```
private void CreateDiagram(AddFlow addflow)
{
    // Create a graphics path which will be used for the custom shape
    GraphicsPath path = new GraphicsPath();
    path.AddArc(0, 0, 15, 15, 180, 90);
    path.AddLine(10, 0, 40, 10);
    path.AddLine(40, 10, 50, 0);
    path.AddLine(50, 0, 80, 40);
    path.AddLine(80, 40, 40, 30);
    path.AddLine(40, 30, 0, 40);
    path.AddLine(0, 40, 10, 20);
    path.CloseFigure();
    path.AddEllipse(10, 10, 10, 10);
    path.FillMode = FillMode.Alternate;

    this.addflow.Dock = DockStyle.Fill;
    this.addflow.AutoScroll = true;
    this.addflow.BackColor = Color.White;
```

```
    this.addflow.PageUnit = GraphicsUnit.Point;

    addflow.NodeModel.FillColor = Color.LightCoral;
    addflow.NodeModel.GraphicsPath = path;
    addflow.NodeModel.ShadowStyle = ShadowStyle.RightBottom;
    addflow.NodeModel.ShadowColor = Color.Silver;
    addflow.NodeModel.ShapeFamily = ShapeFamily.Rectangle;

    Node node1 = new Node(80, 50, 100, 50, null, addflow);
    Node node2 = new Node(260, 160, 70, 70, null, addflow);
    addflow.AddNode(node1);
    addflow.AddNode(node2);
    addflow.AddLink(new Link(node2, node1, null, addflow));
}
```

Notice that it is necessary to add the following line at the beginning of the program:

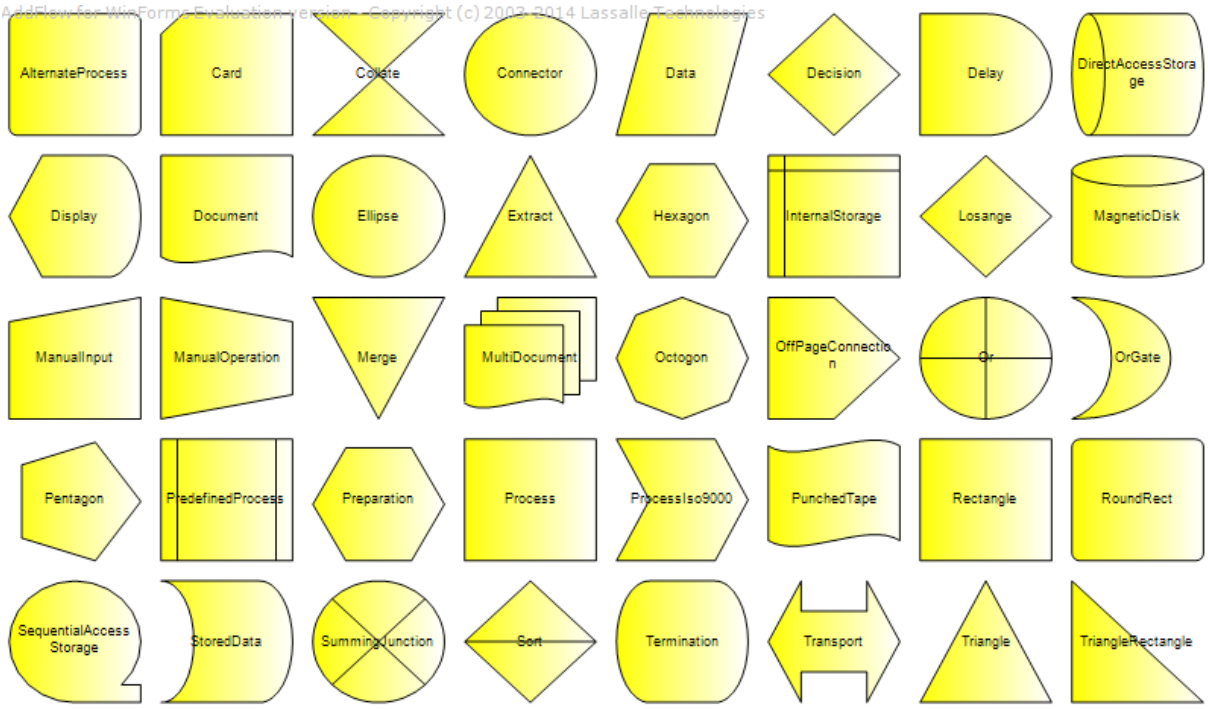
```
using System.Drawing.Drawing2D;
```

If we compile and execute this program, it will create the following diagram:



5.6.4 Predefined shapes

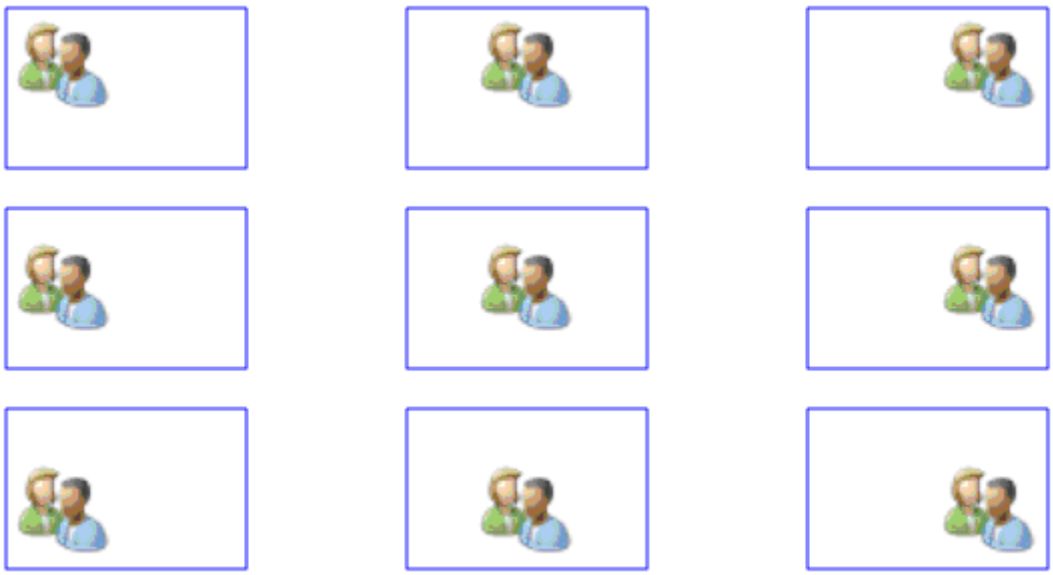
The **PredefinedGeometry** class, defined in the **PredefinedGeometry.cs** file of the **Demo**, provides a list of predefined shapes that be used for ContentItem objects (or link arrows)



5.6.5 ContentItem object image

You can associate an image to a ContentItem object with the **Image** property. Two properties allow placing the image inside the ContentItem object : **ImagePosition** and **ImageMargin**.

This is demonstrated in the **NodeImagePanel.cs** file of the **Demo** sample. In this example, the ImageMargin is set to be equal to 5 at each side.



```
addflow.NodeModel.ImageMargin = new System.Windows.Forms.Padding(5, 5, 5, 5);
```

Of course, the same could be done for a caption as it is also a ContentItem object.

5.7 More informations about Node (Pins)

Only nodes may have pins.

By default, to create a link, you use the 'central pin' of a node.

However, you are not limited to this 'central pin'.

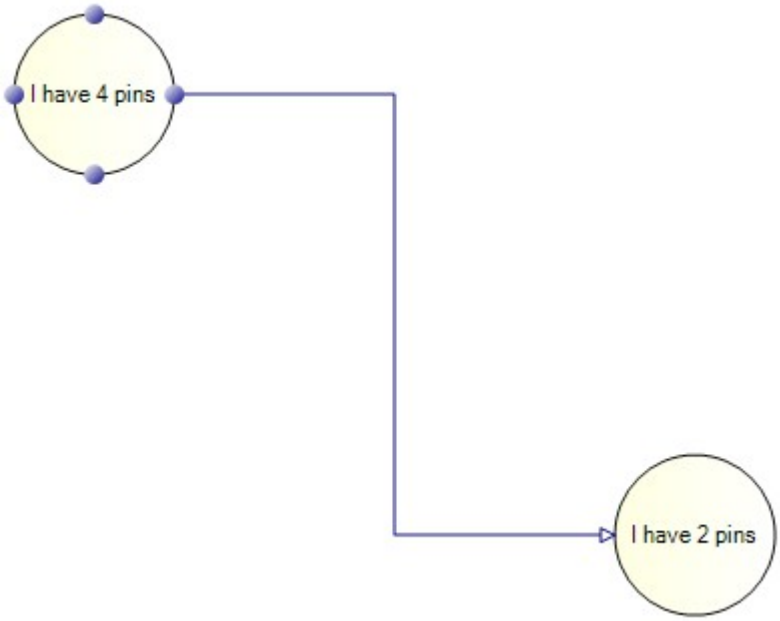
Using the **Pins** property, you may attach a set of pins (also called connectors) to a node. The **Pins** property is an array of points whose coordinates is between 0 and 100. For instance, the following line of code create a set of 4 pins for nodes:

```
addflow.NodeModel.Pins =
    new List<PointF> { new Point(0, 50), new Point(50, 0),
                       new Point(100, 50), new Point(50, 100) };
```

This is demonstrated in the **PinsPanel.cs** file of the **Demo** sample.

```
private void CreateDiagram(AddFlow addflow)
{
    addflow.Dock = DockStyle.Fill;
    addflow.AutoScroll = true;
    addflow.BackColor = SystemColors.Window;
    addflow.PageUnit = GraphicsUnit.Pixel;

    addflow.NodeModel.FillColor = Color.LightYellow;
    addflow.NodeModel.Pins =
        new List<PointF> { new Point(0, 50), new Point(50, 0), new Point(100, 50),
new Point(50, 100) };

    addflow.LinkModel.LineStyle = LineStyle.Orthogonal;
    addflow.LinkModel.DrawColor = Color.Navy;

    // Create 2 nodes. The first two nodes have 4 pins as it is
    // defined by default for every node.
    // However the second node has only 2 pins.
    Node node1 = new Node(100, 30, 80, 80, "I have 4 pins", addflow);
    Node node2 = new Node(400, 250, 80, 80, "I have 2 pins", addflow);
    node2.Pins = new List<PointF> { new Point(0, 50), new Point(100, 50) };
    addflow.AddNode(node1);
    addflow.AddNode(node2);

    // Create one link
    this.AddLink(addflow, node1, node2, 2, 0);
}

Link AddLink(AddFlow addflow, Node org, Node dst, int pinOrgIndex, int pinDstIndex)
{
    Link link = new Link(org, dst, pinOrgIndex, pinDstIndex, "", addflow);
    addflow.AddLink(link);
    return link;
}
```

This function creates the following diagram :



In this example, the line style of the link is 'orthogonal'.

The code used to create such a link is the following:

```
this.AddLink(addflow, node1, node2, 2, 0);
```

Notice the last two parameters that allow setting the index of the origin pin and the index of the destination pin.

5.8 More informations about links

5.8.1 Link colors

Three properties allow setting colors for a node:

- **DrawColor** It is the color of the link line.
- **FillColor** It is the arrow head filling color.
- **TextColor** It is the color of the link text.

5.8.2 Link text

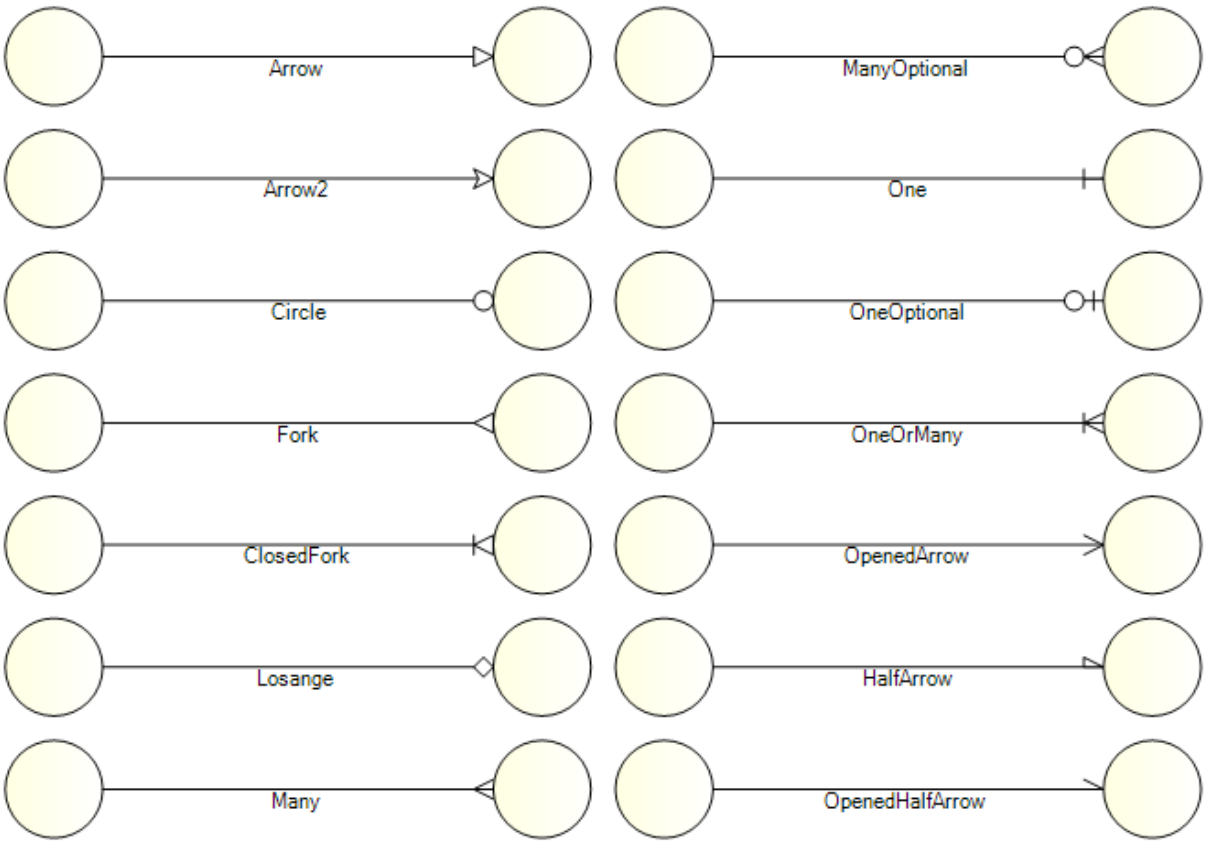
You can associate a text to a node with the **Text** property. The **TextPlacementMode** property allows determining if the text is displayed at the middle point of the link line or at the middle point of the medium segment of the link.

5.8.3 Link arrows

Each link may have an arrow head at its origin if the **IsArrowOrg** property is true, an arrow head at its end if the **IsArrowDst** property is true and an arrow head at the middle of each segment if the **IsArrowMid** property is true.

The arrow head a a default shape provided by AddFlow but it is easy to define custom shapes for arrow heads, using the **ArrowOrg**, **ArrowDst** and **ArrowMid** properties (of type GraphicsPath)

The **PredefinedGeometry** class, defined in the **PredefinedGeometry.cs** file of the **Demo**, provides a list of predefined shapes that be used for link arrow heads). This is demonstrated in the **ArrowPanel.cs** file of the **Demo** sample.

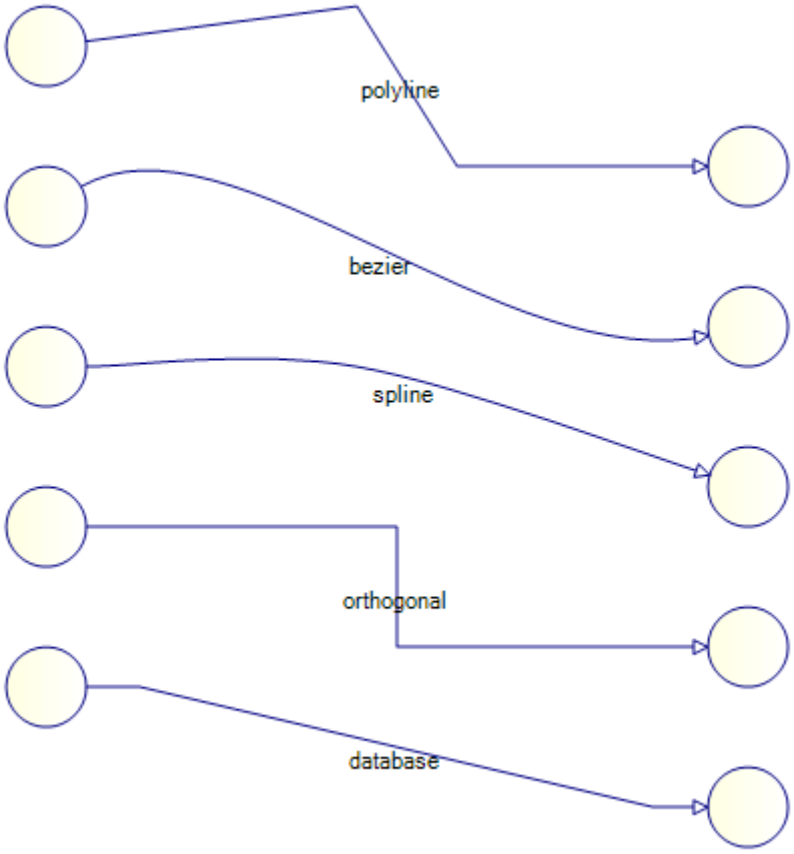


The first arrow is the default one

And of course, as for node shapes, you may define your own style of arrow head shape. The source code in the file PredefinedGeometry.cs may be a starting point for you.

5.8.4 Link line styles

You can define the line style of a link using the **LineStyle** property. This is demonstrated in the **LineStylesPanel.cs** file of the **Demo** sample.



If the line style is 'polyline' or 'spline', you can add as many points as you wish to the link whereas in the other cases, the number of points is fixed: a 'bezier' link or a 'database link' has 4 points. You can move these points but you cannot add new points. The number of points of an orthogonal link is also fixed but at the creation time, depending of the origin and destination pins.

5.9 More informations on Captions

A caption may be used just to display a legend in a diagram. It may also be attached to an AddFlow item. Nodes and links are such items. The property that allows attaching a caption to an item is the **Owner** property. For an AddFlow item, the **Captions** property returns the collection of captions owned by this item. For instance:

```
foreach (Caption caption in node.Captions)
{
    caption.DrawColor = Color.Red;
}
```

WARNING: The Captions collection property is provided only for the AddFlow infrastructure and for enumeration purposes. Don't use it for adding or removing captions. Use instead the Owner property of captions.

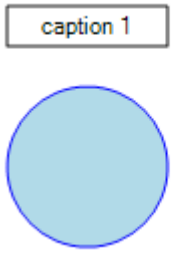
This is demonstrated in the **CaptionsPanel.cs** file of the **Demo** sample.

The following code attach a caption to a node:

```
Node node1 = new Node(40, 100, 80, 80, null, addflow);
addflow.AddNode(node1);
```

```
Caption captionOwnedByNode = new Caption(40, 60, 80, 20, "caption 1",node1,addflow);
addflow.AddCaption(captionOwnedByNode);
```

We obtain the following diagram:



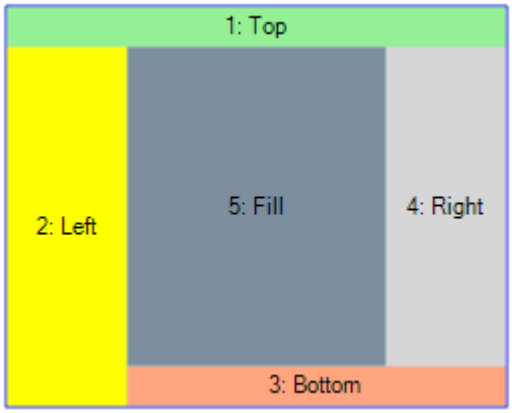
And if you move the node, its caption will follow it.

The **Dock** property of the Caption class allows placing several captions inside a node. It works the same way as the Dock property for controls. For instance, in the following example, the first caption is placed at the top of the owner item because its Dock property is set to DockStyle.Top.

```
// Dock property demo
// Create the "parent" node
Node node2 = new Node(460, 40, 250, 200, null, addflow);
node2.ShapeFamily = ShapeFamily.Rectangle;
addflow.AddNode(node2);

// Create 5 child nodes
Caption child1 = new Caption(50, 130, 90, 20, "1: Top", node2, addflow);
child1.DrawColor = Color.Transparent;
child1.FillColor = Color.LightGreen;
child1.GradientColor = Color.LightGreen;
child1.Dock = DockStyle.Top;
child1.IsSelectable = false;
child1.IsHitTestVisible = false;

Caption child2 = new Caption(50, 160, 60, 20, "2: Left", node2, addflow);
child2.DrawColor = Color.Transparent;
child2.FillColor = Color.Yellow;
child2.GradientColor = Color.Yellow;
child2.Dock = DockStyle.Left;
child2.IsSelectable = false;
child2.IsHitTestVisible = false;

Caption child3 = new Caption(50, 190, 60, 20, "3: Bottom", node2, addflow);
child3.DrawColor = Color.Transparent;
child3.FillColor = Color.LightSalmon;
child3.GradientColor = Color.LightSalmon;
child3.Dock = DockStyle.Bottom;
child3.IsSelectable = false;
child3.IsHitTestVisible = false;

Caption child4 = new Caption(50, 220, 60, 20, "4: Right", node2, addflow);
child4.DrawColor = Color.Transparent;
child4.FillColor = Color.LightGray;
child4.GradientColor = Color.LightGray;
child4.Dock = DockStyle.Right;
child4.IsSelectable = false;
child4.IsHitTestVisible = false;

Caption child5 = new Caption(50, 250, 90, 20, "5: Fill", node2, addflow);
child5.DrawColor = Color.Transparent;
```

```
child5.FillColor = Color.LightSlateGray;
child5.GradientColor = Color.LightSlateGray;
child5.Dock = DockStyle.Fill;
child5.IsSelectable = false;
child5.IsHitTestVisible = false;

// Add the items to the diagram.
addflow.AddCaption(child1);
addflow.AddCaption(child2);
addflow.AddCaption(child3);
addflow.AddCaption(child4);
addflow.AddCaption(child5);
```

We obtain the following diagram:

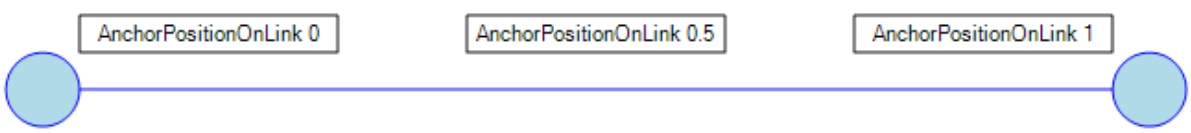


Au you can see, the captions have been placed automatically in the owner node.

Captions may also be owned by a link. The **AnchorPositionOnLink** property allows defining the position of the caption near the link. It is a value between 0 and 1. If it is 0, then the caption is placed near the origin node of the link. If it is 1, then the caption is placed near the destination node of the link. If it is for instance 0.5, then the caption is placed near the middle of the link.

```
Node node3 = new Node(60, 400, 40, 40, null, addflow);
addflow.AddNode(node3);

Node node4 = new Node(660, 400, 40, 40, null, addflow);
addflow.AddNode(node4);

Link link = new Link(node3, node4, null, addflow);
addflow.AddLink(link);

Caption captionOwnedByLink1 = new Caption(100, 380, 140, 20, "AnchorPositionOnLink
0", link, addflow);
captionOwnedByLink1.AnchorPositionOnLink = 0f;
addflow.AddCaption(captionOwnedByLink1);

Caption captionOwnedByLink2 = new Caption(310, 380, 140, 20, "AnchorPositionOnLink
0.5", link, addflow);
captionOwnedByLink2.AnchorPositionOnLink = 0.5f;
addflow.AddCaption(captionOwnedByLink2);

Caption captionOwnedByLink3 = new Caption(520, 380, 140, 20, "AnchorPositionOnLink
1", link, addflow);
captionOwnedByLink3.AnchorPositionOnLink = 1f;
addflow.AddCaption(captionOwnedByLink3);
```

We obtain the following diagram:



5.10 OwnerDraw property

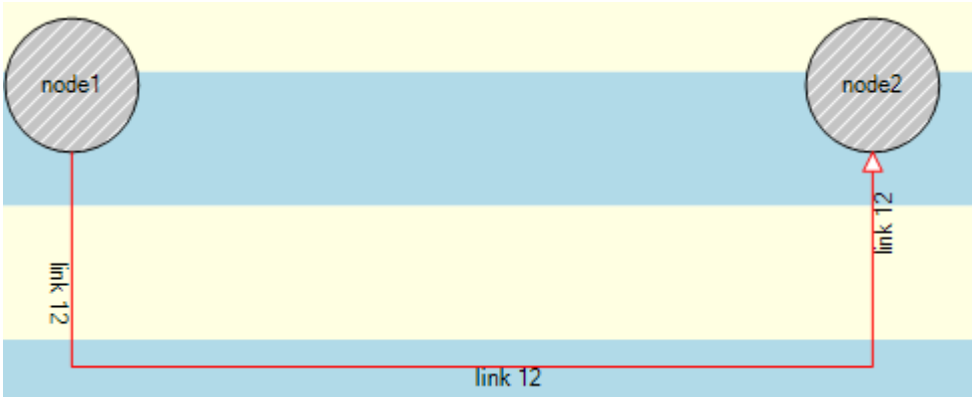
Even though there are many properties offering a lot of control over the appearance of nodes and links (colors, font, styles, etc), sometimes that is not enough. You may want to use a hatched background inside a node, or draw some custom graphics (for instance swim lines) directly in the background of the AddFlow control.

In these cases, you can use the “OwnerDraw” properties and events to gain total control over how each item is drawn. You can customize the drawing of a node, a link or the entire AddFlow control.

Object	Property	Event	Event parameter
AddFlow	IsOwnerDraw	DiagramOwnerDraw	DiagramOwnerDrawEventArgs
Node	IsOwnerDraw	NodeOwnerDraw	NodeOwnerDrawEventArgs
Link	IsOwnerDraw	LinkOwnerDraw	LinkOwnerDrawEventArgs

If the **IsOwnerDraw** property is true, then the corresponding event is fired each time the object needs to be redrawn, giving the possibility to replace the default drawing made by AddFlow by a custom drawing. Each event parameter class contains a **Flags** property which specifies how the drawing is made.

This is demonstrated in the **OwnerDrawPanel.cs** file of the **Demo** sample. The following diagram is created :



Notice the swim lines in the background, the nodes with hatched brush and the link text which is displayed near each segment.

For instance, following is the code used to draw a node with a hatched background:

```
private void node_OwnerDraw(object sender, NodeOwnerDrawEventArgs e)
{
    Graphics grfx = e.Graphics;
    Node node = e.Node;

    // Save the graphics state because it does not belong to us
    // (although in this case, it not necessary because we do not
    // alter the Graphics state)
    GraphicsState gs = grfx.Save();

    HatchBrush hBrush1 = new HatchBrush(HatchStyle.BackwardDiagonal,
                                        Color.White, Color.Silver);
    grfx.FillEllipse(hBrush1, node.Rect);

    // Restore the graphics state
    grfx.Restore(gs);

    // Tell AddFlow not to fill the node (because this is already done)
    e.Flags = e.Flags & ~(NodeDrawFlags.Fill);
}
```

Of course, your program should have the following line of code:

```
addflow.NodeOwnerDraw += new AddFlow.NodeOwnerDrawEventHandler(node_OwnerDraw);
```

5.11 Diagram navigation

AddFlow provides a set of properties and methods to navigate in a diagram (“Network traversals”). Notice that the majority of the properties and methods described here are demonstrated in the **NavigPanel.cs** file of the **Demo** sample.

The **Items** property of the AddFlow control allows accessing every item (nodes, links, captions) of a diagram. For instance:

```
foreach (Item item in addflow.Items)
{
  item.DrawColor = Color.Red;
}
```

If you are interested only by nodes, you may write:

```
foreach (Item item in addflow.Items)
{
  if (item is Node)
  {
    item.DrawColor = Color.Red;
  }
}
```

or, using LINQ to select in an array all the nodes of the canvas :

```
var nodes = addflow.Items.OfType<Node>().ToArray();
foreach (Node node in nodes)
{
    node.DrawColor = Color.Red;
}
```

The **SelectedItems** property of the AddFlow control allows accessing every selected item (nodes, links, captions) of a diagram. For instance:

```
foreach (Item item in addflow.SelectedItems)
{
  item.DrawColor = Color.Red;
}
```

The **Links** property of the Node object allows accessing every link that come to a node or leave it. For instance:

```
foreach (Link link in node.Links)
{
  link.DrawColor = Color.Red;
}
```

To access only the selected links :

```
foreach (Link link in node.Links)
{
    if (link.Org == node)
    {
        link.DrawColor = Color.Red;
    }
}
```

(or you may use LINQ)

The **GetLinkedNode** method of the Node object returns the node connected via a given link.

```
node2 = node1.GetLinkedNode(link);
```

The **Org** property of the Link object returns/sets the reference of the origin node of the link.

The **Dst** property of the Link object returns/sets the reference of the destination node of the link.

There is also the **Captions** property of AddFlow items. For instance:

```
foreach (Caption caption in node.Captions)
{
    caption.DrawColor = Color.Red;
}
```

5.12 Selection of items

5.12.1Interactive selection

You can select an item interactively by clicking it with the mouse.

You can also select several items interactively by clicking them with the mouse and simultaneously pressing the shift or control key.

Or you can select items with a selection rectangle, if the **MouseSelection** property is set to MouseSelection.Selection. In this last case, you bring the mouse cursor into the AddFlow control, press the left button, move the mouse and release the left button. All nodes or links or captions partly inside the selection rectangle are selected. Then you can unselect some items by clicking them with the mouse and simultaneously pressing the shift or control key. You can select them again by using the same method.

TIP: How to select interactively a link with the mouse?

If the link is made of one or several segments, then if you want to select it with the mouse, you have just to click near one of its segments. If the link is a Bezier or a Spline curve, then you have just to click near the curve.

5.12.2Selection handles

If the **IsSelectionHandleDisplayed** property is set to false, then the selection handles used to emphasize the selected items are not displayed. This may be useful if you wish to use your own way to show the selected items as ine **GenealogyPanel**.cs file of the **Demo** sample.

5.12.3Programmatic selection: ISelectable interface

Both Node and Link classes implement the **ISelectable** interface:

- An item (node, caption or link) can be selected either interactively, either programmatically using its **IsSelected** property, for instance:

```
node.IsSelected = true;
link.IsSelected = true;
caption.IsSelected = true;
```

- If the **IsSelectable** property of an item is false, then it is no more possible to select it.

5.12.4SelectedItems collection

The **SelectedItems** collection property of AddFlow allows getting each selected item. For instance:

```
// Make each selected nodes red
IEnumerable<Node> selnodes = this.addflow.SelectedItems.OfType<Node>();
foreach (Node node in selnodes)
{
    node.BorderBrush = new SolidColorBrush(Colors.Red);
}
```

WARNING: there is a difference with previous versions. In this version, there is not any notion of a “current” (or “active”) item and therefore there is not any property like SelectedNode, SelectedLink or SelectedItem. When several items (nodes or links) are selected, you cannot designate one that could be the current one.

5.12.5Selection event

The **SelectionChanged** event is fired each time the selection status of an item is changed. However, you can avoid that by setting the **CanSendSelectionChangedEvent** property to false.

5.12.6Hit Testing

You can also know what object is under the mouse with the **HitItem** property that returns the reference of the item under the mouse. If several objects are under the mouse, the returned object is the one that is at the top of the Z-order list. You may change this order with the **ZOrder** property of the Item object.

If the **IsHitTestVisible** property of an item is false, then this item cannot be hit tested.

Instead of using the **HitItem** property, you may use the **GetItemAt** method.

5.13 Some other information about drawing

AddFlow provides also some properties and methods that control the general aspect of a diagram:

Extent	Returns the size of the diagram.
PageScale	Returns/sets a value that determines the scaling used to display the graph.
PageUnit	Returns/sets a value that determines the unit of measure used to display the graph.
Zoom	Returns/sets the zooming factor.
ZoomPoint	Zoom the diagram and center it on a point
ZoomRectangle	A method allowing zooming a rectangular portion of the diagram

TIP: How to autofit the diagram; i.e. how to adjust the zoom to its maximum while still keeping all the shapes (nodes, links etc.) in view?

The **ZoomRectangle** method and the **Extent** property should allow implementing this feature:

```
RectangleF rc = new RectangleF(new PointF(0, 0), addflow.Extent);
addflow.ZoomRectangle(rc);
```

5.14 Serialization

- AddFlow does not provide any serialization feature (a design choice)
- The **Demo** sample shows how to provide a XML serialization (see the file **xmlflow.cs**).
- Node and caption custom shapes and link arrow custom shapes are not saved.
- The xmlflow.cs module allows also loading diagrams created with previous versions. However, there are some restrictions: children nodes are just loaded as nodes but their relationship with a parent node is ignored.
- In a future version, JSON serialization could also be provided.

5.15 Printing a diagram

AddFlow itself does not offer any printing feature. However, the printing and the print previewing of AddFlow diagrams can be demonstrated in the **Demo** sample (see the file **prnflow.cs**)

5.16 Exporting the diagram

5.16.1 The Render method

The Render method can be used to display the diagram in any GDI+ drawing area. This method could be used for instance to implement a “bird view” window.

5.16.2 Metafile support

You can export an AddFlow diagram as a Metafile with the **ExportMetafile** method. The exported image can be saved in a file, copied in a picture box or in the clipboard. This method is demonstrated in the **Demo** sample.

5.17 Data customization

5.17.1 Tag property

The **Tag** property allows associating an object to an AddFlow item (node, link, caption)

5.17.2 Property bag

The property bag allows extending the functionality of AddFlow items by adding new properties. You can do that with the **Properties** property.

For instance, if you wish to add a new property “Author” to a node and assign the value "Alice" to this property, you can do that using a single line of code:

```
node.Properties["Author"].Value = "Alice";
```

In this example, this property is a string but it could be any kind of objects.

The Property bag feature is demonstrated in the **PropertyBagPanel.cs** file of the **Demo** sample.

5.17.3 Derivation of Node, Caption and Link classes

The **DeriveNodePanel.cs** file of the **Demo** sample shows how to create a new class MyNode, derived from the Node class.

5.17.3.1 The derived class

The class MyNode, derived from the Node class, contains two new string properties “Author” and “Comment”.

```
internal class MyNode : Node
{
    public MyNode(Node defnode, string author, string comment)
        : base(defnode)
    {
        this.Author = author;
        this.Comment = comment;
    }

    public string Author { get; set; }

    public string Comment { get; set; }
}
```

5.17.3.2 Interactive creation of a derived node

The user creates interactively (with the mouse) a MyNode object exactly as he would to create a node. To implement that, we use the **BeforeAddNode** event which is fired just before a node is created interactively (or also programmaticaly if the InteractiveEventsOnly property is false).

In the handler of this event, the following tasks are done:

- Cancel the node creation
- Instead of creating a node, we create a MyNode object. However, as the MyNode class inherits form the Node class, our MyNode object is also a Node object.
- The MyNode object is placed at the same place as the node whose creation has been aborted.
- The MyNode object is then added to the diagram.

The code is the following:

```
private void AddFlow1_BeforeAddNode(object sender,BeforeAddNodeEventArgs e)
{
    e.Cancel.Cancel = true;
    MyNode mynode = new MyNode(addflow.NodeModel, null, null);
    mynode.Location = e.Location;
    mynode.Size = e.Size;
    addflow.AddNode(mynode);
    Mynode.IsSelected = true;
}
```

5.17.3.3 Add Custom data

In the DeriveNode sample, when the user double click on a node and if this node is in fact a MyNode object (which is always the case in our DeriveNode sample), a dialog box is displayed to allow entering the MyNode object custom data (here the "Author" and the Comment" strings).

```
private void EnterCustomData()
{
    if (this.addflow.SelectedItems.Count > 0 &&
        this.addflow.SelectedItems[0] is Node)
    {
        Node node = this.addflow.SelectedItems[0] as Node;
        if (node is MyNode)
        {
            MyNode mynode = (MyNode)node;
            NodeData nd = new NodeData();
            nd.MyNode = mynode;
            if (nd.ShowDialog() == System.Windows.Forms.DialogResult.OK)
            {
                this.addflow.TaskManager.SubmitTask(new MyTask(mynode));
                mynode.Author = nd.TextBoxAuthor.Text;
                mynode.Comment = nd.textBoxComment.Text;
            }
        }
    }
}
```

(Notice the call to **SubmitTask** to include the custom data change in the undo/redo buffer. You can find the code of the MyTask class in the **DeriveNodePanel.cs** file of the **Demo** sample)

6 Avanced topics

6.1 Undo/Redo

6.1.1 General features

AddFlow has a property named **TaskManager** of type TaskManager that provides a powerful multilevel Undo/Redo feature. The history length is limited only by available memory. However, you can limit it yourself with the **UndoLimit** property of the TaskManager class. You can also enable/disable the undo/redo with the **CanUndoRedo** property of AddFlow.

6.1.2 Updating the user interface

Some properties and methods allow you to properly update the user interface. The **CanUndo** and **CanRedo** methods will tell you if there is something to undo or redo and therefore will allow you to grey out the menu options. The **RedoCode** and **UndoCode** properties return a code that describes the action waiting to be redone or undone. This will allow your application to give descriptions of the actions on the undo and redo history.

6.1.3 Grouping basic actions

Every basic action has a code. However, the **BeginAction** and **EndAction** methods allow you to define a group of actions and to assign a code to this group. This is useful if for instance, in your application, the user can open a dialog box allowing changing several properties of a node (for instance, its text, its shape and its filling color). You will certainly wish to allow the user to undo these 3 basic actions in one time.

Notice that you can also stop recording actions with the **SkipUndo** method and also clear the Undo/Redo buffer with the **ResetUndoRedo** buffer.

Another interesting method is the **AddToLastAction** method. For instance, it allows grouping some actions with the last recorded action or group of actions.

Notice that you have to call the **EndAction** to terminate the group of actions.

6.1.4 Undo/Redo customization

The undo/redo can be customized. For that, you have to create a custom Task class by deriving the Task class and then you can insert it in the undo list with the **SubmitTask** method.

6.1.5 What can be undone and redone?

The rule is the following: every action that changes a diagram can be undone or redone. This includes actions like moving or resizing nodes or stretching links.

However, making a selection does not change the document so you will not be able to undo a selection. Changing properties of the AddFlow control (zoom, grid, default filling color, etc) does not change the document too. Therefore, it will not be possible to undo these actions.

6.1.6 Undo/Redo API

AddToLastAction	Add the following actions in the last group of actions
BeginAction	Start a group of actions that can be undone in one time.
CanRedo	Indicates if there is an action that can be redone.
CanUndo	Indicates if there is an action that can be undone.
CanUndoRedo	Determines whether undo/redo is allowed.
Clear	Clears the undo/redo buffer.
EndAction	Terminate a group of actions that can be undone in one time.
Redo	Redo, if possible, the last action.
RedoCode	Returns the code of the next redoable action.
RedoItem	Returns the item involved in the next redoable action
SkipUndo	Determines whether the following actions are recorded in the undo manager.
SubmitTask	Submit a task (or action) that can be undone and redone.
RemoveLastTask	Remove the last task that has been added in the undo list.
Undo	Undo, if possible, the last action.
UndoCode	Returns the code of the next undoable action.
UndoItem	Returns the item involved in the next undoable action.
UndoLimit	Sets and returns the number of undo commands that can be performed.

6.2 Performance tuning

6.2.1 BeginUpdate and EndUpdate methods

To maintain performance while items are added to the AddFlow control, you should call the **BeginUpdate** method. The BeginUpdate method prevents the control from painting until the **EndUpdate** method is called.

Also, between the calls to BeginUpdate and EndUpdate, the size of the diagram is not updated. It is updated only when the EndUpdate method is called. If you need the size of the diagram before the call to EndUpdate, you can use the **GetDiagramSize** method.

The two methods BeginUpdate and EndUpdate are often used in the **Demo** sample.

6.2.2 IsQuadtree property

If you have a very big diagram (100000 items), you may find that it takes time to select an item. This can be accelerated if you set the **IsQuadtree** property. If this property is set, the AddFlow items are managed in a Quadtree data structure that allows finding then quickly.

6.2.3 Other tips for better performances

- Disable the Undo/Redo! If you are creating programmatically a big diagram (as in the **StressPanel** example of the **Demo**), it is also important to disable the undo/redo with the **CanUndoRedo** property.
- Do not create link jumps! You should use jumps only for small diagrams (less than 200 nodes) because the algorithm used to find intersections requires considerable computational resources. By default however, link jumps are not created (See the **JumpSize** property of links).

6.3 Graph Algorithms

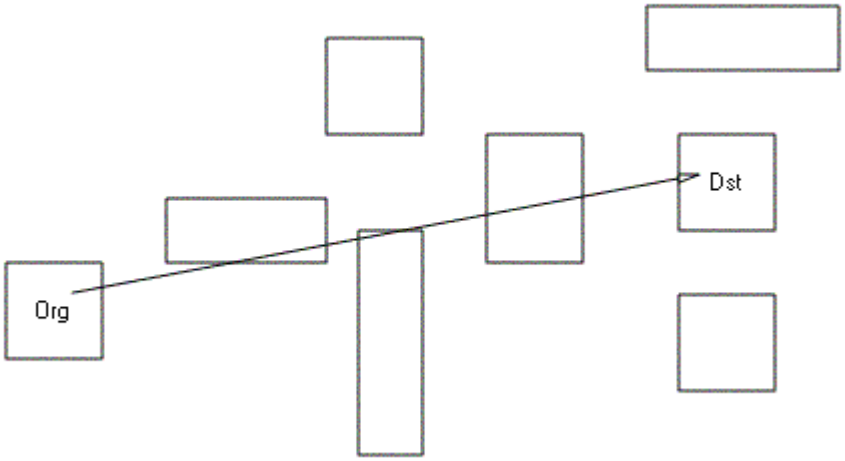
The **GraphAlgoPanel.cs** file of the **Demo** sample demonstrates how to use AddFlow to implement some well-known graph algorithms: connectivity algorithms, spanning tree algorithms, etc... The algorithms are implemented in the file **AlgoFlow.cs.** The **Tag** property is used to extend the Node and Link classes. For instance, a **IsVisited** property is added to nodes and links and a **Cost** property is added to links.

6.4 Link auto-routing

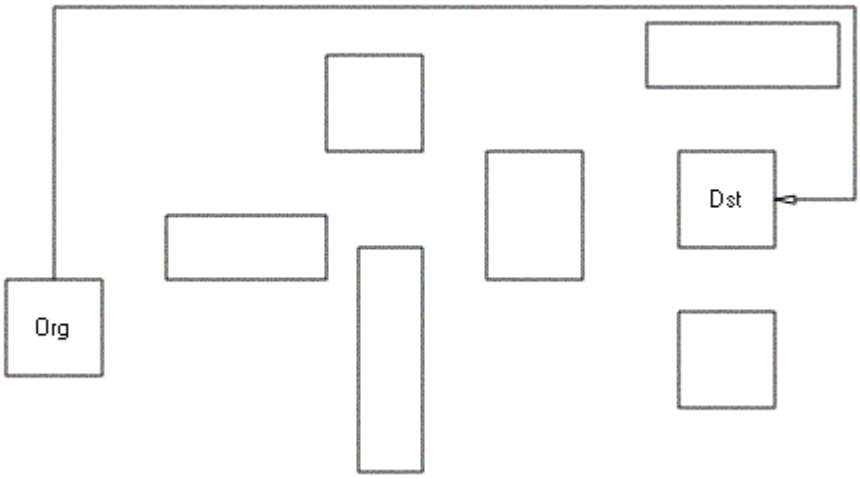
6.4.1 Introduction

The **RouteFlowPanel.cs** file of the **Demo** sample demonstrates how to do link auto-routing. It uses the Autoroute class provided in the file **RouteFlow.cs**.

By default, when the user creates a link between 2 nodes, the link is drawn as a straight line which may traverse other nodes, as in the following example.



The RouteFlow algorithm allows finding a route between the other nodes which are considered as obstacles, as illustrated in the following picture.



6.4.2 Method

RouteFlow uses the following rules:

1) Orthogonality rule

The path is composed of orthogonal segments. A new segment is always orthogonal to the previous one.

2) Distance rule (we try to decrease the distance towards the destination node) When we move in a segment:

- If the distance towards the destination node is increasing, then we stop the current segment and create a new segment allowing decreasing the distance towards the destination node.
- Else we continue until:
    - Either we hit an obstacle (another node)
    - Either we reach the X or Y coordinates of the destination.

3) Obstacle rule

- If we hit an obstacle, we backtrack to the previous segment and we continue to move.
- If this is not possible, we move back one step and we create a new segment.

4) Start rule

The first segment is orthogonal to a side of the origin node rectangle. Several methods to select the starting side are provided via the **StartingMethod** property.

6.4.3 Code sample

The following code is all you need to do to perform a route finding for a link. This code should be included in the handler of the AfterAddLink event of AddFlow.

```
Autorouting route = new Autorouting();
route.SendStepEvent = this.sendStepEvent;
route.Grain = this.grain;
route.MinDistance = this.minDistance;
route.StartingMethod = this.startingMethod;
```

In the sample, a little animation may be launched, using the **Step** event. If the **SendStepEvent** property is true, the **Step** event is sent at each step of the algorithm.

6.4.4 Limitations

There are some limitations. The auto-routing is performed only if:

- The link is not reflexive
- The line style is Polyline
- The IsAdjustDst and IsAdjustOrg properties are true.

And finally, in some cases (pathological diagrams), the auto-routing algorithm may fail to provide a correct solution. In such cases, it will just draw only one segment joining the origin and destination nodes.

Remark: By default, a node is considered as an obstacle but you may alter this behavior using the custom property **IsNotObstacle.** For instance:

```
node.Properties["IsNotObstacle"].Value = "true";
```

6.5 AddFlow capabilities

Following properties allow to set capabilities for an AddFlow control and therefore to customize it. For instance, if you wish to allow only one link between two nodes, you have just to unset the **CanMultiLink** property.

BezierSelectionLineColor	Returns/sets the drawing color of the lines used for selected bezier links.
CursorSetting	Determines whether default cursors are displayed.
CanChangeDst	Determines whether the user can interactively change the destination of a link.
CanChangeOrg	Determines whether the user can interactively change the origin of a link.
CanDragScroll	Determines whether drag scrolling is allowed or not.
CanDrawNode	Determines whether interactive creation of nodes is allowed or not.
CanDrawCaption	Determines whether interactive creation of captions is allowed or not.
CanDrawLink	Determines whether interactive creation of links is allowed or not.
CanFireError	Determines if the error event is fired or not.
CanMoveItems	Determines whether interactive dragging of items is allowed or not.
CanMultiLink	Determines whether you can create several links between two nodes.
CanMultiSelect	Determines whether multiselection of nodes is allowed or not.
CanReflexLink	Determines whether interactive creation of reflexive links is allowed or not.
CanShowTooltips	Determines whether the tooltips are displayed or not.
CanShowJumps	Determines whether jumps are displayed at the intersection of links.
CanSizeItems	Determines whether interaction resizing of items is allowed or not.
CanStretchLink	Determines whether interactive stretching of links is allowed or not.
CanUndoRedo	Determines whether undo/redo is allowed.
LinkModel	Defines the default property values for links.
CaptionModel	Defines the default property values for captions.
NodeModel	Defines the default property values for nodes.
IsSelectionHandleDisplayed	Determines whether the handles used for selection are displayed or not.
GridSnap	Determines whether nodes are aligned on the grid.
GridDraw	Determines whether the grid is displayed or not.

GridStyle	Returns/sets the grid style.
GridSize	Returns/sets the grid size.
GridColor	Returns/sets the grid color.
HandleDrawColor	Returns/sets a value which defines the color used for the pen of the selection handles.
HandleColor1	Returns/sets a value which defines the first color used for the brush of the selection handles.
HandleColor2	Returns/sets a value which defines the second color used for the brush of the selection handles.
HandleSize	Returns/sets the size of the selection handles.
LinkHandleSize	Defines the size of the linking handle at the center of selected node.
LinkSelectionAreaWidth	Determines the width of the area where the user has to click to select a link.
IsOwnerDraw	Determines whether you want to provide custom drawing for the diagram.
PinDrawColor	Returns/sets a value which defines the color used for the pen of the node pins.
PinColor1	Returns/sets a value which defines the first color used for the brush of the node pins.
PinColor2	Returns/sets a value which defines the second color used for the brush of the node pins.
PinSize	Returns/sets a value which defines the size of the node pins.
RemovePointDistance	Returns/sets the angle that causes a link point to be removed when stretching a link.
SelectionHandleSize	Defines the size of the selection handles of the selected node.

6.6 Customization

As you have seen in previous paragraphs, there are several ways to customize AddFlow and you can customize the AddFlow behaviour, the AddFlow drawings and also the data associated to nodes and links.

a) Behaviour customization

- **AddFlow capabilities**.
- **AddFlow class derivation**.

b) Drawing customization

- **Custom shapes**. You can customize the AddFlow drawings by using the possibility to create custom shapes for nodes and links.
- **OwnerDraw property**.You can also customize the AddFlow drawings using the OwnerDraw property.

c) Data customization

There are 3 ways to associate custom data to a node or link:

- **Tag property**. (ithis method is used for instance in the **AlgoFlow**.**cs** file)
- **Property bag**. (this method is used for instance in the **RouteFlow**.**cs** file)
- **Derivation of Node, Caption or Link classes**. (It is used in the **DeriveNodePanel**.**cs** file)

7 Automatic Graph LayoutAutomatic Graph Layout

7.1 Introduction

The primary purpose of an automatic graph layout feature is to offer a way to display graphs or flow charts in a reasonable manner, following some aesthetic rules.

AddFlow does not provide directly any automatic graph layout feature. However, we propose **LayoutFlow which provides** a set of 5 graph layout algorithms:

- Hierarchic layout
- Orthogonal layout
- Symmetric layout
- Series Parallel layout
- Tree layout

Each of these graph layout algorithms performs a layout on a graph. Performing a layout automatically positions its nodes (also called vertices) and links (also called edges).

Typically, you can first create your nodes and links inside AddFlow, using the AddFlow API, giving each node a random or a (0,0) position. Then you call the layout method of the graph layout control of your choice. This method will position the nodes and the links in a **reasonable** manner in the AddFlow control, following some aesthetic rules that depend on the chosen control (hierarchical, symmetric, orthogonal...).

Remarks

- Currently, LayoutFlow is an AddFlow extension and you cannot use it without AddFlow. If you just want to perform a layout on a graph without displaying them in an AddFlow control (for instance because you have already a way to display the diagram), then you can use both a hidden AddFlow control and LayoutFlow to do that. In such a case, AddFlow is just used to store the logical structure of the graph and to retrieve via its API, the resulting positions of its nodes and links.
- The **Demo** sample shows how to use each graph layout component.
- Reflexive links are not taken into account by layout algorithms. Reflexive links are just translated to follow their origin (and also destination) node.

TIP: How to manage so that the layout algorithm applies only to a subset of the graph?

You know we provide a mechanism to add custom properties to each item. It is the “property bag”. LayoutFlow is using such a property: **IsExcludedFromLayout**. Only the nodes and links whose IsExcludedFromLayout property is false are involved in each layout. This will allow you to make the layout algorithm to ignore some nodes or links. By default, the IsExcludedFromLayout property is false.

For instance, the following line is excluding the node “node” from the layout.

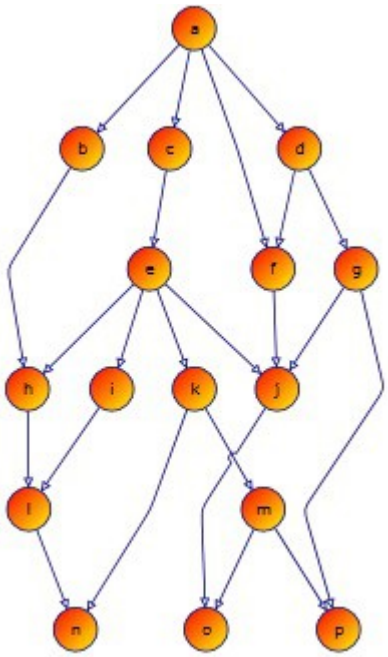
```
node.Properties["IsExcludedFromLayout"].Value = "true";
```

This is demonstrated in the HierarchicPanel.cs file of the **Demo** sample

7.2 Hierarchic layout

7.2.1 Purpose

This algorithm performs a hierarchical layout on a graph. The hierarchical layout arranges vertices in horizontal layers. The order of the nodes on the layers is chosen so that the number of crossings is kept as small as possible.



- Hierarchic layout -

7.2.2 Code example

The following code is all you need to do to perform a hierarchical layout:

```
LayoutFlow.TreeLayout(this.addflow,
              50, // Sets the distance between adjacent levels
              50, // Sets the distance between adjacent nodes
              Orientation.North,
              new Size(20, 20), // Margin
              0); // No limitation in the number of nodes in a level
```

This code supposes that you have a form containing an AddFlow control. You create the graph in the AddFlow control, either interactively, either programmatically (in this case, giving each node a random position or a (0,0) position). Then you apply the layout to this graph. And each bode will be placed at a reasonable position.

7.2.3 Limitation

It works with any graph, connected or not.

7.2.4 Side Effect

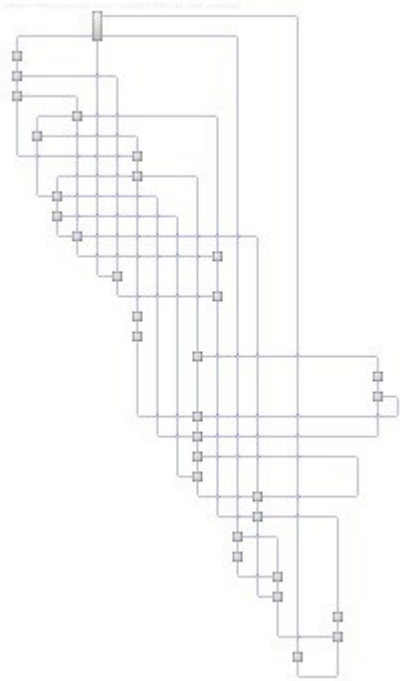
After the layout execution:

- the line style of the links is Polyline
- the IsAdjustOrg property of links is set to true.
- the IsAdjustDst property of links is set to true.

7.3 Orthogonal layout

7.3.1 Purpose

This algorithm performs an orthogonal layout on a graph. The layout is orthogonal since it produces an orthogonal drawing where each link is drawn as a polygonal chain of alternating horizontal and vertical segments. The algorithm used is the Biedl and Kant algorithm.



- Orthogonal layout -

7.3.2 Code example

The following code is all you need to do to perform an orthogonal layout:

```
LayoutFlow.OrthogonalGridLayout(this.addflow,
       Orientation.North,
       new Size(40, 40), // The horizontal and vertical grid size
       nodeSizeRatio, // The node size (in percentage of the grid size)
       new Size(20, 20)); // Margin
```

7.3.3 Limitation

It works with any graph, connected or not. Note however that this algorithm is making generous use of space and the resulting layout is good only with small graphs.

7.3.4 Side Effect

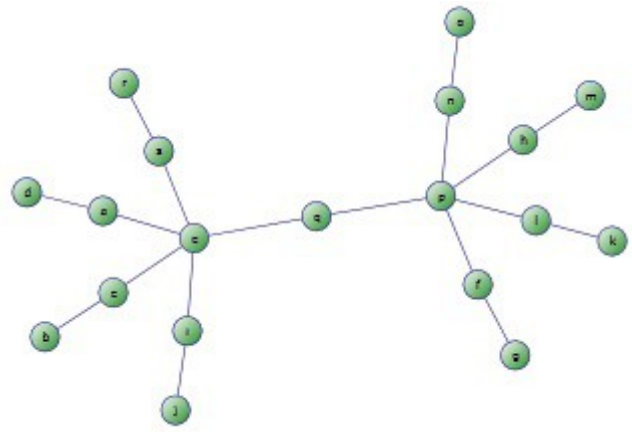
After the layout execution:

- the size of the nodes is changed. If the graph is a graph of maximum degree four, then each node has the same size (determined by the **GridSize** property). If the degree of a node is higher than four, then the height of the node is expanded.
- the line style of the links is Polyline.
- the IsAdjustOrg property of links is set to false.
- the IsAdjustDst property of links is set to false.

7.4 Force Directed (Symmetric) layout

7.4.1 Purpose

This algorithm performs a symmetric layout on a graph. This layout produces a high degree of symmetry and is particularly useful for undirected graphs, where the directions of the links are not important. It is using a force-directed algorithm (the GEM method of Frick, Ludwig and Mehldau) where a graph is viewed as a system of bodies with forces acting between the bodies.



- Symmetric layout -

7.4.2 Code example

The following code is all you need to do to perform a symmetric layout on a graph:

```
LayoutFlow.SymmetricLayout(this.addflow,
                    50,                 // Sets the distance between nodes
                    new Size(20, 20)); // Margin
```

7.4.3 Limitation

It works with any graph, connected or not. However, it is recommended to work only with small graphs (less than 1000 nodes) because it is using a force-directed method and force-directed methods are using considerable computational resources.

7.4.4 Side Effect

After the layout execution:

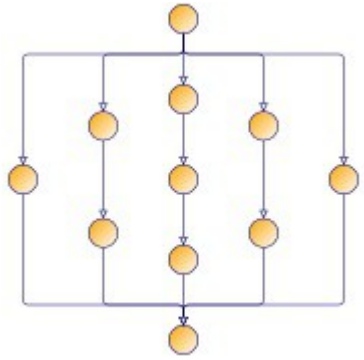
- the line style of the links is Polyline and each link is composed of only one segment.
- the IsAdjustOrg property of links is set to true.
- the IsAdjustDst property of links is set to true.

7.5 Series-parallel layout

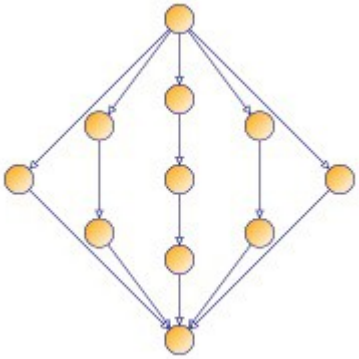
7.5.1 Purpose

This algorithm performs a series-parallel layout on a graph. The SP layout applies only to a specific subset of graphs: series-parallel digraph (more precisely, a set of series-parallel diagraphs). A series-parallel digraph is defined recursively as follows.
A digraph consisting of two nodes, a source s and a sink t joined by a single link is a series-parallel digraph.
If G1 and G2 are series-parallel digraphs, so are the digraphs constructed by each of the following operations:
- the parallel composition: identify the source of G1 with the source of G2 and the sink of G1 with the sink of G2.
- the series composition: identify the sink of G1 with the source of G2.

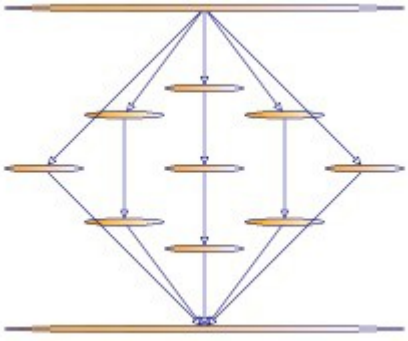
We use an algorithm (described in the book "Drawing Graphs" Michael Kaufmann - Dorothea Wagner) that allows drawing series-parallel digraphs with as much symmetry as possible.



- SP layout: DrawingStyle = BusOrthogonalDrawing



- SP layout: DrawingStyle = StraightLine -



- SP layout: DrawingStyle = VisibilityDrawing -

7.5.2 Code example

The following code is all you need to do to perform a series-parallel layout on a graph:

```
LayoutFlow.SeriesParallelLayout(this.addflow,
       DrawingStyle.BusOrthogonalDrawing,
       Orientation.North,
         80,                // Sets the distance between adjacent levels
         80,                // Sets the distance between adjacent nodes
         new Size(30, 30),  // The vertex size
         new Size(20, 20)); // Margins
```

If the graph is not a set of series-parallel digraph, an exception is generated.

7.5.3 Limitation

The layout applies only to a specific subset of graphs: series-parallel digraphs. One of the requirements is that this diagram has only one starting node and only one ending node. However, it is not actually a limitation. If, for instance, the number of ending nodes is greater than one, then a workaround is to create a dummy node and create a link from each ending node to this dummy node, then execute the layout and then delete the dummy node (which causes all the dummy links to be deleted too).

7.5.4 Side Effect

After the layout execution:

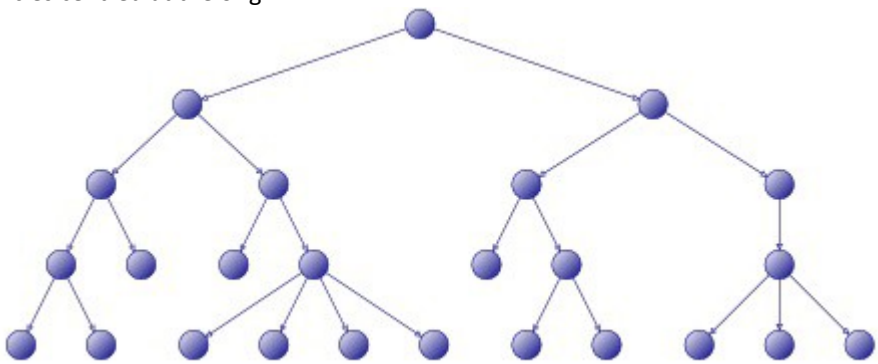
- the line style of the links is Polyline. Moreover, if the DrawingStyle property is not BusOrthogonalDrawing, then each link is composed of only one segment.
- the IsAdjustOrg property of links is set to true.
- the IsAdjustDst property of links is set to true.

7.6 Tree layout

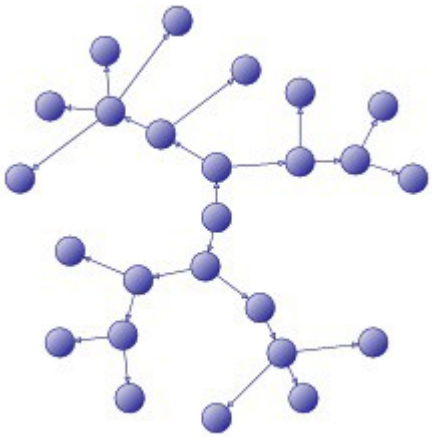
7.6.1 Purpose

This algorithm performs a tree layout on a graph. This layout applies only to a specific subset of graphs: rooted trees. In such a graph, no node may have more than one parent. It offers two drawing styles (**DrawingStyle** property).

- If the DrawingStyle is **Layered**, then the drawing of the tree occupies as little space as possible while satisfying certain aesthetics: nodes at the same level of the tree are placed on the same line and a parent is centred over its children.
- If the DrawingStyle is **Radial**, then the root of the tree is placed at the origin and the layers are concentric circles centred at the origin.



- Tree layout: DrawingStyle = Layered -



- Tree layout: DrawingStyle = Radial -

7.6.2 Code example

The following code is all you need to do to perform a tree layout on a graph:

```
LayoutFlow.TreeLayout(this.addflow,
                      50, // Layer distance
                      50, // Vertex distance
                      DrawingStyle.Layered,
                      Orientation.North,
                      new Size(20, 20)); // Margin
```

If the graph is not a forest of rooted trees, an exception is generated.

7.6.3 Limitation

The layout applies only to a specific subset of graphs: rooted trees. More precisely, the layout applies to forests (sets of rooted trees).

7.6.4 Side Effect

After the layout execution:

- the line style of the links is Polyline
- the IsAdjustOrg property of links is set to true.
- the IsAdjustDst property of links is set to true.

8 Conversion guide

8.1 Introduction

AddFlow For Winforms 2015 is **not compatible** with previous versions. Almost everything has changed. We are going to list those changes in the first paragraph then give a table of conversion.

8.2 Main changes

Some features have been removed. Some are placed outside AddFlow. Some are presented in adifferent way. Some are added.

8.2.1 Node parent/child relationship

The parent/child relationship between nodes has been removed. It is replaced by a new type of objects: captions. A caption can be owned by any AddFlow items, including captions.

8.2.2 Drag Frame

In previous version, a drag frame was displayed on the node border when the node was selected. This drag frame was used to drag the node. This feature has been removed. Now to drag a node, you have just to click on it.

8.2.3 Serialization

AddFlow does not provide any serialization feature. However, we provide instead in the **Demo** sample the source code allowing to save/load a diagram in XML format: it is implemented in the file **XMLFlow.cs** (sour code provided). This module allows also to loading diagrams created with previous versions but there are some restrictions: children nodes are just loaded as nodes but their relationship with a parent node is ignored.

8.2.4 Collections

The collections defined in AddFlow must not be used directly except for enumeration purpose (foreach …). Instead you have to use some methods provided by AddFlow to add or remove nodes or links , points.

For instance:

instead of writing:

```
node = addflow.Nodes.Add(100, 1100, 50, 50);
```

you have use the AddNode method.

instead of writing:

```
link = node1.OutLinks.Add(node2);
```

you have use the AddLink method.

instead of writing:

```
link.Points.Add(new PointF(100, 150));
```

you have to write:

```
link.AddPoint(new PointF(100, 150));
```

Notice also that some collection have been removed.

- the Nodes collection

- the OutLinks and InLinks collections of the Node object

We can live without those collections, especially with the use of the LINQ technology. For instance, if you want the collection of nodes, you have just to wrote:

```
var nodes = addflow.Items.OfType<Node>().ToArray();
```

8.2.5 Selection of items

In this new version, there is not any notion of a “current” (or “active”) item and therefore there is not any property like SelectedNode, SelectedLink or SelectedItem. When several items (nodes or links) are selected, you cannot designate one that could be the current one.

If for instance, you select a node and wish to change its color, you can write:

```
if (this.addflow.SelectedItems.Count > 0 && this.addflow.SelectedItems[0] is Node)
{
    Node node = this.addflow.SelectedItems[0] as Node;
    if (node != null)
    {
        node.FillColor = Color.Red;
    }
}
```

8.2.6 Images

Now, you may associate an image to a node using just the **Image** property. The ImageIndex property is removed.

8.2.7 Connection features

The ConnectedItems and CycleMode properties have been removed. This kind of features is now placed in the **GraphAlgo.cs** file of the **Demo** sample.

8.2.8 Pins

This is a new major feature of AddFlow. A Pin allows creating a link from the pin of a node to the pin of another node.

8.2.9 Captions

This is new feature. A Caption is like a label or a note. It is a new type of object that allows displaying a text or an image and that can be owned by any item.

8.2.10 Orthogonals links

It is a new line style for links.

8.2.11 No more grouped properties (except one)

The Shape, Shadow, Line, Arrow, Grid and Zoom classes are removed and replaced by their memeber properties.

For instance the Grid property is replaced by the 5 following properties : **GridSnap, GridDraw, GridSize, GridColor** and **GridStyle**.

The exception is the new **TaskManager** class that manages the undo/redo feature.

8.2.12 Renaming

The naming of many properties and methods has changed. For instance the boolean properties (except the capabilities properties like CanStretchLink and also the GridDraw and GridSnap properties) start with «Is» prefix. I like this practice used in WPF or Silverlight.

For instance the OutLinkable property is renamed IsOutLinkable.

8.3 Table of conversion

8.3.1 AddFlow properties

Previous versions	AddFlow for Winforms 2015
AntiAliasing	Removed
CanLabelEdit	CanEditContentItem
CanRedo	TaskManager.CanRedo
CanUndo	TaskManager.CanUndo
CanUndoRedo	TaskManager.CanUndoRedo
CycleMode	Removed
DefLinkProp	Renamed → LinkModel
DefNodeProp	Renamed → NodeModel
DisplayDragFrame	Removed
DisplayHandles	Renamed → IsSelectionHandleDisplayed
Grid	Replaced by 5 properties: GridColor, GridDraw, GridSnap, GridSize, GridStyle
Images	Removed
InteractiveAction	Replaced by the following boolean properties: IsCreatingLink, IsCreatingNode, IsCreatingCaption, IsStretchingLink, IsMovingNode, IsResizingNode, IsSelecting, IsPaning, IsZooming
InteractiveEventsOnly	Renamed → IsInteractiveEventsOnly
IsChanged	Works only if undo/redo enabled
JumpSize	It is now a property of the Link object
LinkCreationMode	Removed
LinkHandleSize	Removed
MultiSel	Renamed → CanMultiSelect
Nodes	Removed
PageGrid	Removed
PointedArea	Renamed → HitArea
PointedItem	Renamed → HitItem
RedoCode	TaskManager.RedoCode
RedoItem	TaskManager.RedoItem
RemovePointAngle	Renamed → RemovePointDistance
RoundedCornerSize	It is now a property of the Link object
SelectedItem	Removed
SelectionHandleSize	Renamed → HandleSize
SendSelectionChangeEvent	Renamed → IsSelectionChangeEventFired
ShowTooltips	Renamed → CanShowTooltips
SkipUndo	TaskManager.SkipUndo

UndoCode	TaskManager.UndoCode
UndoItem	TaskManager.UndoItem
UndoSize	TaskManager.UndoSize
Zoom	The type has changed. It is now a float number instead of Size.

8.3.2 AddFlow methods

Previous versions	**AddFlow for Winforms 2015**
BeginAction	TaskManager.BeginAction
AddToLastAction	TaskManager.AddToLastAction
CreateLink	Removed
EndAction	TaskManager.EndAction
GetSchema	Removed
ReadXml	Removed
Redo	TaskManager.Redo
ResetUndoRedo	TaskManager.Clear
SetChangedFlag	Works only if undo/redo enabled
SubmitTask	TaskManager.SubmitTask
Undo	TaskManager.Undo
WriteXml	Removed

8.3.3 AddFlow events

All the events dealing with XML serialization are removed.

8.3.4 Node properties

Previous versions	**AddFlow for Winforms 2015**
Alignment	Renamed → TextPosition
AttachmentStyle	Removed
AutoSize	Removed
Children	Captions
ConnectedItems	Removed
Dock	Dock is now aproperty of the caption class
DrawWidth	Renamed → Thickness
Gradient	Removed
Hidden	Removed
HighlightChildren	IsCaptionsHilighted
ImageIndex	Removed (use Image property instead)
ImageLocation	Renamed → ImageMargin
Index	Removed
InLinkable	Renamed → IsInLinkable

InLinks	Removed
LabelEdit	IsEditable
Logical	Removed (use PropertyBag instead)
OutLinkable	Renamed → IsOutLinkable
OutLinks	Removed
OwnerDraw	Renamed → IsOwnerDraw
Parent	Owner (property of Caption class)
Rect	Renamed → Bounds
RemoveChildren	Removed
Selectable	Renamed → IsSelectable
Selected	Renamed → IsSelected
Shadow	Removed. Use instead the ShadowSize, ShadowStyle and ShadowColor properties.
Shape	Removed
Transparent	Removed (use a transparent color instead)
Trimming	Removed
Url	Removed
XMoveable	Renamed → IsXMoveable
XSizeable	Renamed → IsXSizeable
YMoveable	Renamed → IsYMoveable
YSizeable	Renamed → IsYSizeable

8.3.5 Node methods

GetPath	Removed
GetSchema	Removed
ReadXml	Removed
ReadXmlProperties	Removed
Remove	Removed
WriteXml	Removed

8.3.6 Link properties

Previous versions	**AddFlow for Winforms 2015**
AdjustDst	Renamed → IsAdjustDst
AdjustOrg	Renamed → IsAdjustOrg
ArrowDst	Its type has changed. It is now a GraphicsPath
ArrowMid	Its type has changed. It is now a GraphicsPath
ArrowOrg	Its type has changed. It is now a GraphicsPath
Children	Captions

ConnectionStyleDst	Removed
ConnectionStyleOrg	Removed
CustomEndCap	Removed
CustomStartCap	Removed
EndCap	Removed
Hidden	Removed
HighlightChildren	IsCaptionsHilighted
Jump	Removed. Use the JumpSize property instead.
Line	Removed. Use instead directly the LineStyle property.
Logical	Removed
OrientedText	Renamed → IsOrientedText
OwnerDraw	Renamed → IsOwnerDraw
Points	Don't use this property (except for enumeration) which is only provided for the AddFlow infrastructure. To manipulate the collection of link points, you should use instead the methods AddPoint, RemovePoints, ClearPoints, CountPoints, GetPoint and SetPoint.
RemoveChildren	Removed
Rigid	Removed
Selectable	Renamed → IsSelectable
Selected	Renamed → IsSelected
Shadow	Removed. Use instead the ShadowSize, ShadowStyle and ShadowColor properties.
StartCap	Removed
Stretchable	Renamed → IsStretchable

8.3.7 Link methods

Previous versions	**AddFlow for Winforms 2015**
GetPath	Removed
GetSchema	Removed
ReadXml	Removed
ReadXmlProperties	Removed
Remove	Removed
Reverse	Removed
WriteXml	Removed
WriteXmlProperties	Removed