Copyright (c) 2009-2011 Lassalle Technologies. All Rights Reserved

AddFlow for WPF V1.3 Tutorial

February 2011

Lassalle Technologies

http://www.lassalle.com

Copyright (c) 2009-2011 Lassalle Technologies. All Rights Reserved

CONTENTS

1 Introduction 5

2 Last Version enhancements.....6

2.1 Version 1.3 enhancements.....6

2.2 Version 1.2 enhancements.....6

2.3 Version 1.1 enhancements.....6

- 2.3.1 New Features.....6
- 2.3.2 Compatibility.....7

3 Getting Started.....8

3.1 Installation.....8

3.2 AddFlow extensions.....8

3.3 AddFlow free extensions.....8

3.4 Licensing.....9

- 3.4.1 Type of licenses.....9
- 3.4.2 How it works?.....9

4 Interactive creation of a diagram.....12

4.1 Overview.....12

4.2 Create a diagram interactively.....12

- 4.2.1 Draw a node.....12
- 4.2.2 Draw a link.....12
- 4.2.3 Stretch a link.....14
- 4.2.4 Draw a reflexive link.....15
- 4.2.5 Multiselection.....15
- 4.2.6 Node rotation.....16
- 4.2.7 Change properties of a node or a link.....17
- 4.2.8 Add a text to a node.....17
- 4.2.9 Adjust the link origin and destination points.....17
- 4.2.10 Change the destination or the origin node of a link.....18

5 Programmatic creation of a diagram.....20

5.1 Overview.....20

5.2 Diagram creation.....21

- 5.2.1 Our first program.....21
- 5.2.2 Node Properties.....24
- 5.2.3 Link Properties.....25
- 5.2.4 Changing property values.....25
- 5.2.5 Default property values.....26
- 5.2.6 Stretching the links.....27

5.3 How to define the node geometry.....29

5.4 Displaying the node content.....30

- 5.4.1 Node text.....30
- 5.4.2 Node image.....30

5.5 Displaying a shadow for a node or a link.....31

Copyright (c) 2009-2011 Lassalle Technologies. All Rights Reserved

5.6 Displaying link intersections....32

5.7 Selection of items....33

- 5.7.1 Interactive selection....33
- 5.7.2 Programmatic selection....33
- 5.7.3 SelectedItems collection....33
- 5.7.4 Selection events....34

5.8 Diagram navigation....34

5.9 Zooming....34

5.10 Serialization....34

5.11 Printing a diagram....35

5.12 Exporting a diagram in XAML....35

5.13 Customizing the user interface....37

- 5.13.1 Customizing the handles....37
- 5.13.2 Customizing the connection of links to a node....39
  - 5.13.2.1 Generalities....39
  - 5.13.2.2 Customizing pins....40
  - 5.13.2.3 Customizing the Connector....40

6 Avanced topics....43

6.1 Undo/Redo....43

- 6.1.1 General features....43
- 6.1.2 Updating the user interface....43
- 6.1.3 Grouping basic actions....43
- 6.1.4 What can be undone and redone?....43
- 6.1.5 Undo/Redo customization....44
- 6.1.6 Undo/Redo API....45

6.2 Automatic Graph Layout....47

- 6.2.1 Hierarchic layout....48
  - 6.2.1.1 Purpose....48
  - 6.2.1.2 Code example....48
  - 6.2.1.3 Limitation....48
  - 6.2.1.4 Side Effect....48
- 6.2.2 Orthogonal layout....49
  - 6.2.2.1 Purpose....49
  - 6.2.2.2 Code example....49
  - 6.2.2.3 Limitation....49
  - 6.2.2.4 Side Effect....50
- 6.2.3 Symmetric layout....50
  - 6.2.3.1 Purpose....50
  - 6.2.3.2 Code example....50
  - 6.2.3.3 Limitation....50
  - 6.2.3.4 Side Effect....51
- 6.2.4 Series-parallel layout....51
  - 6.2.4.1 Purpose....51
  - 6.2.4.2 Code example....52
  - 6.2.4.3 Limitation....53
  - 6.2.4.4 Side Effect....53
- 6.2.5 Tree layout....53
  - 6.2.5.1 Purpose....53
  - 6.2.5.2 Code example....54
  - 6.2.5.3 Limitation....54
  - 6.2.5.4 Side Effect....54

6.3 Data Customization....55

- 6.3.1 Framework's Tag property....55
- 6.3.2 Attached properties....55
- 6.3.3 Derivation of Node and Link classes....55

Copyright (c) 2009-2011 Lassalle Technologies. All Rights Reserved

6.4 Conversion guide from previous versions.....56

- 6.4.1 Introduction.....56
  - 6.4.1.1 Removed Features.....56
  - 6.4.1.2 Features that work differently.....56
  - 6.4.1.3 Renamed features.....56
- 6.4.2 Conversion guide from AddFlow ActiveX.....56
  - 6.4.2.1 AddFlow properties.....56
  - 6.4.2.2 AddFlow methods.....59
  - 6.4.2.3 AddFlow events.....59
  - 6.4.2.4 Node properties.....60
  - 6.4.2.5 Node methods.....61
  - 6.4.2.6 Link properties.....61
  - 6.4.2.7 Link methods.....62
- 6.4.3 Conversion guide from AddFlow for .NET.....63
  - 6.4.3.1 AddFlow properties.....63
  - 6.4.3.2 AddFlow methods.....63
  - 6.4.3.3 AddFlow events.....64
  - 6.4.3.4 Node properties.....65
  - 6.4.3.5 Node methods.....66
  - 6.4.3.6 Link properties.....66
  - 6.4.3.7 Link methods.....67

1 Introduction

AddFlow for WPF is a general purpose Flowcharting/Diagramming WPF component, which lets you quickly build flowchart-enabled WPF applications.

AddFlow for WPF 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, …

It has been created with **VS 2010** and **.NET Framework 4** on a 64 bits machine.

Purpose of this tutorial

This tutorial provides information on:

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

Who should use this tutorial?

This guide is intended for application programmers using the WPF platform to build WPF applications.

Samples

AddFlow for WPF is installed with one demo sample written in C#: DemoFlow

2 Last Version enhancements

2.1 Version 1.3 enhancements

1. **Rotation feature**.

Nodes can be rotated. If the **CanRotateNode** property of AddFlow is true (which is the case by default) and if the **IsRotatable** property of a node is true (which is the case by default), a node can be rotated programmatically, using the node **RotationAngle** property or interactively, using a handle placed at the right of the node.

You may change the look of this rotation handle using the **RotateHandleStyle** property.

When rotating nodes interactively, the **SelectedNodesRotating** event is fired and the IsRotatingNode property of AddFlow is true.

2. **Interactive zoom**

This release adds the possibility to zoom the diagram interactively with the mouse if the **MouseSelection** property of the AddFlow control is set to **MouseSelection.Zoom**.

3. **Interactive selection enhancement**

A new property to accelerate the selection of items inside a rectangle drawn with the the mouse: **CanSelectOnMouseMove**. It is true by default. If false, the selection is made only at the MouseUp event (see for instance the “Stress” example in the demo)

4. **New styles**

It adds also two properties **SelectionBoxStyle** and **ZoomingBoxStyle** allowing defining the style of the rectangle used when selection items or zooming interactively with the mouse (demonstrated in the “Stress” example in the demo)

2.2 Version 1.2 enhancements

It has been created with **VS 2010** and **.NET Framework 4** on a 64 bits machine.

2.3 Version 1.1 enhancements

2.3.1 New Features

This new release provides the following new features:

1. **Link intersections**

   AddFlow uses a powerful algorithm to find intersections between the link segments. If the **CanShowJumps** property of the AddFlow control is true then jumps will be displayed at the intersection of this link with other links.

2. **Panning mode**

If you set the MouseSelection property to MouseSelection.Pan, you enter in panning mode: you can scroll the diagram directly with the mouse (don't need to use the scrollbars)

3. **Connectors and pins**

   You can know customize the shape of the pins used to connect create the links. And you can define for each node a connector object that contains several pins.

2.3.2 Compatibility

The following changes have been made to be consistent with the Silverlight version.

Two properties are renamed:

- the **ResizeThumbStyle** property is renamed **ResizeHandleStyle**
- the **StretchThumbStyle** property is renamed **StretchHandleStyle**

Two methods are now obsolete: **BeginUpdate** and **EndUpdate.** You don't need to use them anymore.

The **IsOrgAdjustable** and **IsDstAdjustable** link properties are removed and are replaced by the node properties:**OutConnectionMode** and **InConnectionMode.** It is the node that decides how the links are connected to itself. It is no more a property of the link.

3 Getting Started

3.1 Installation

The AddFlow for WPF 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 WPF installation folder, it creates 3 subdirectories: Bin, Doc and Demo:

- The **Bin** subdirectory contains the assemblies: DLLs, demo executable. The list of the DLLs is given in the following parapagraph.
- 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 DemoFlow sample that demonstrates AddFlow for WPF. It contains also the source code of the **XMLFlow** and **AlgoFlow** free extensions.

3.2 AddFlow extensions

Following is the list of the AddFlow for WPF extensions, including AddFlow itself. All these assemblies are installed with AddFlow for WPF.

Assembly	**Description**
Lassalle.WPF.Flow.dll	The AddFlow for WPF control
Lassalle.WPF.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 WPF 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 can purchase a source code license. A source code license agreement will have to be signed in such a case. Notice that all these assemblies are written in C#.

3.3 AddFlow free extensions

These extensions can be used freely. Their source code is either installed with AddFlow in the Demo directory, either provided if you purchase the product. We do not provide support for these free extensions.

Assembly	Description	Source code
Lassalle.AlgoFlow.dll	The AlgoFlow free extension which provides some graph algorithms	Installed with AddFlow
Lassalle.WPF.Flow.XML.dll	The XMLFlow free extension which provides XML serialization of AddFlow diagrams	Installed with AddFlow
Lassalle.WPF.Geometries.dll	This static class allows creating some predefined geometries to be used for nodes or links	Provided if you purchase AddFlow (and if you ask for it)

3.4 Licensing

3.4.1 Type of licenses

Notice that you can purchase either:

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

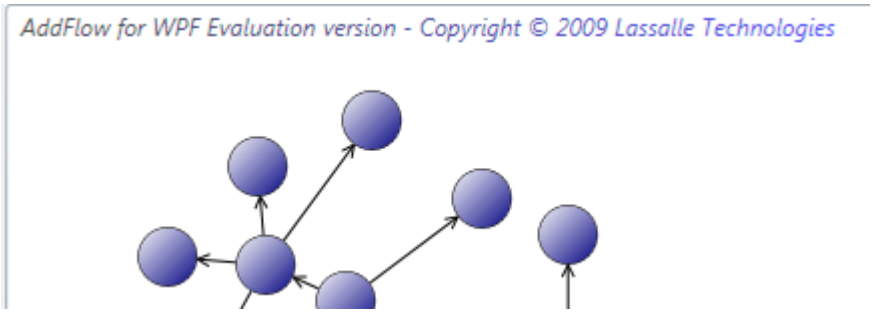
3.4.2 How it works?

The evaluation version

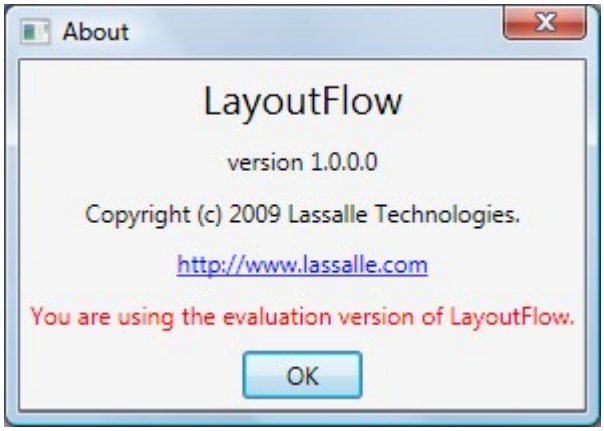
When you install AddFlow for WPF, you install in fact an evaluation version of AddFlow. (And you install also an evaluation version of LayoutFlow)

If you generate ("compile") an application that uses this evaluation version of AddFlow for WPF, 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).

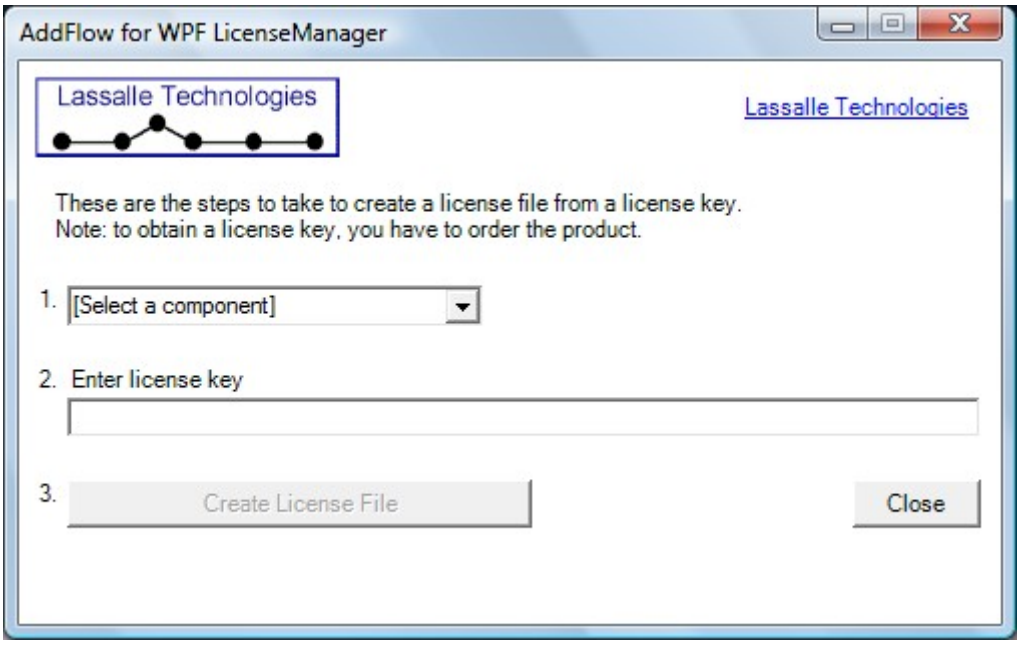
Now, it depends of the type of license you have purchased.

- If you have just purchased a Standard license of AddFlow for WPF, 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 WPF, no nag screen will be displayed if you execute a graph layout methods.

The LicenseManager program

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



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

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

Example

After having installed AddFlow for WPF, load the DemoFlow 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 WPF 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 WPF, 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 **DemoFlow** 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.WPF.Flow.AddFlow, Lassalle.WPF.Flow, Version=1.3.0.0, Culture=neutral,
PublicKeyToken=8e9784929b011e15
```

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.

4 Interactive creation of a diagram

4.1 Overview

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 rotation 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
- In place edition for nodes

It supports also the scrolling of diagrams and the use of 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 **CanMoveNode** properties.

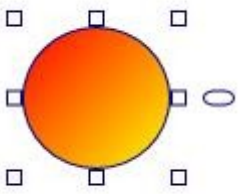
And a set of methods and properties allow implementing a powerful Undo/Redo feature.

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.

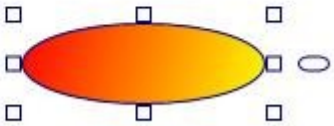
The 8 handles allow **resizing the node** (the handle at the right allows rotating it). If you want to **move the node,** you bring the mouse cursor into the node (but not in the center), press the left button, move the mouse and release the left button.



4.2.2 Draw a link

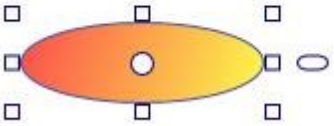
Draw a second node.



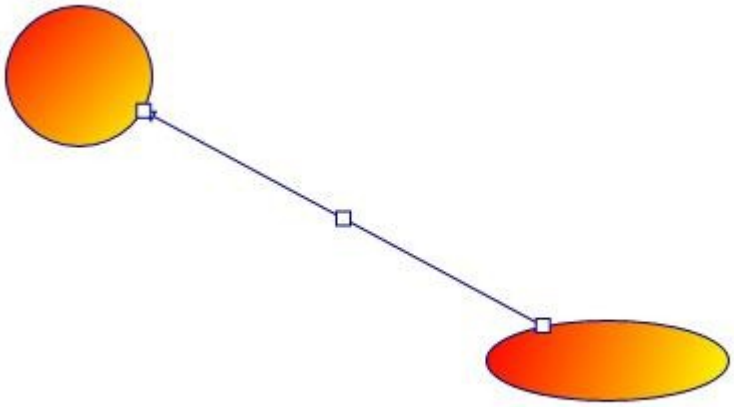


Then bring the mouse cursor above the second node. A small circle handle is then displayed at the center of the selected node.





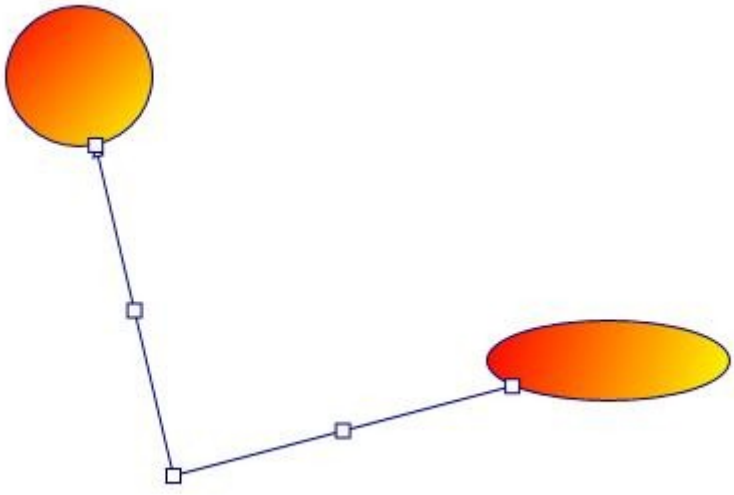
Bring the mouse over this small circle handle, press the left button, move the mouse towards the other node. When the mouse cursor is into the other node, release the left button. The link has been created. And it is selected: 3 handles are displayed in the link.



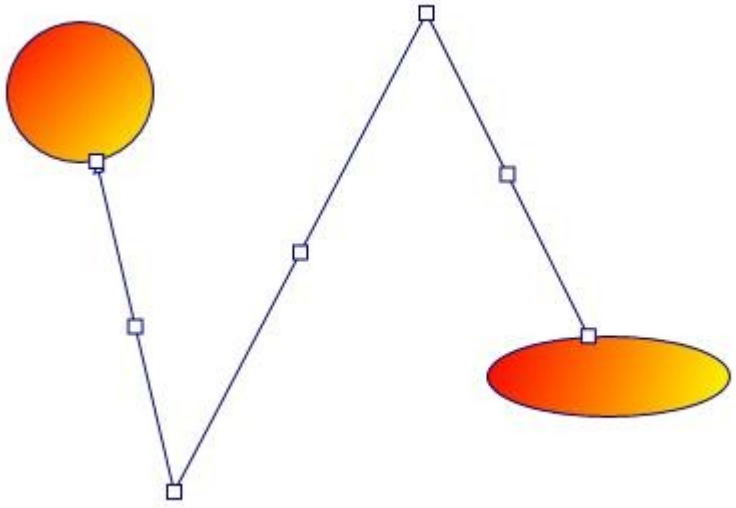
As you can see, the link stretching handles are also displayed as little rectangles. By default, those handles are small rectangles as for the nodes above. But we can change the style of those handles. The DemoFlow sample provided with AddFlow shows many distinct ways to display the node resizing handles and the link stretching handles.

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).

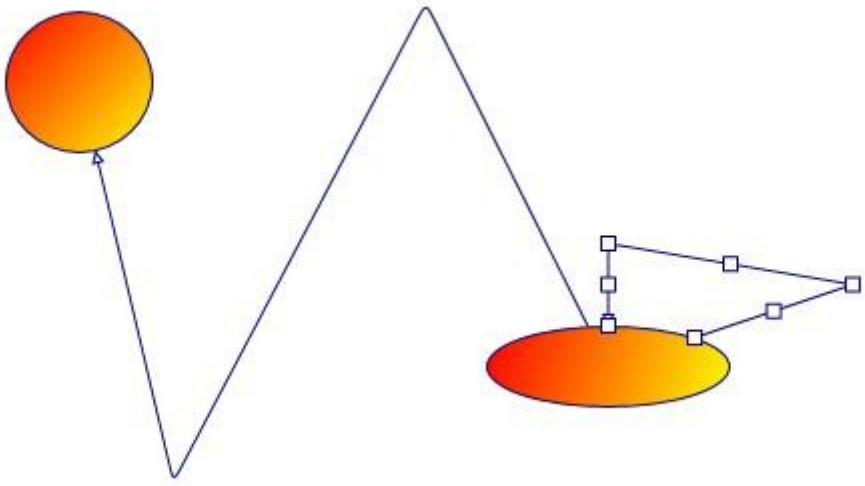


Create another segment



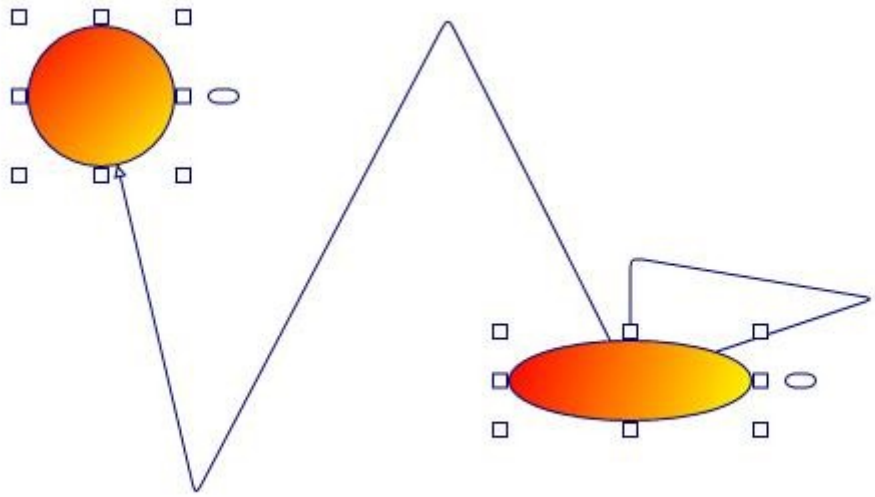
4.2.4 Draw a reflexive link

Select a node by clicking on it. Then bring the mouse cursor above the small diamond handle at the center of 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.

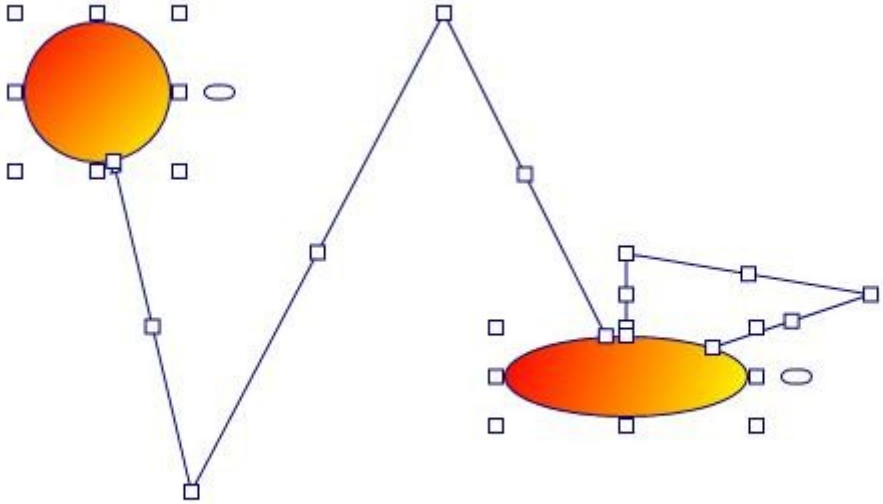


4.2.5 Multiselection

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



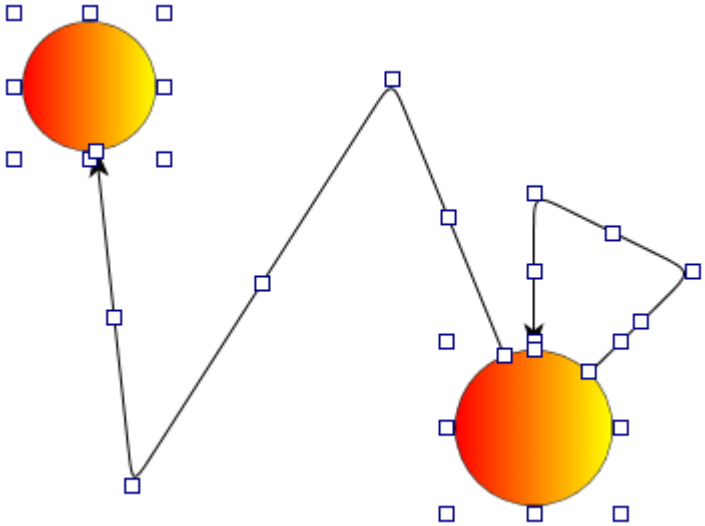
You can also select links or nodes and links.



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 Node rotation

Bring the mouse cursor into the handle iplaced at the right of a node, press the left button, move the mouse and release the left button. You have rotated the node.

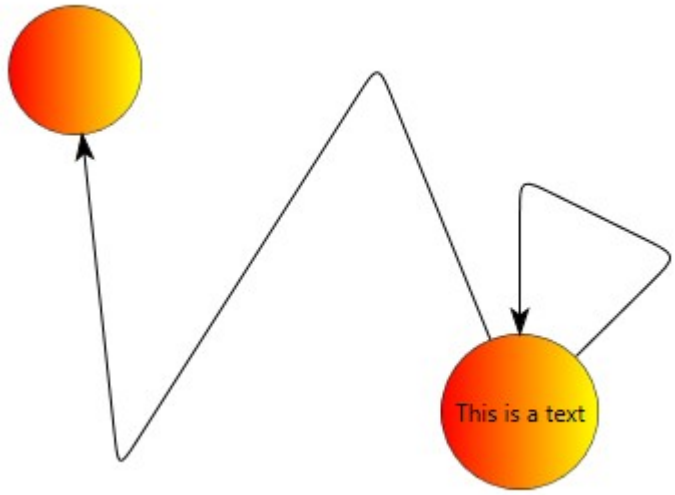


4.2.7 Change properties of a node or a link

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.

4.2.8 Add a text to a node

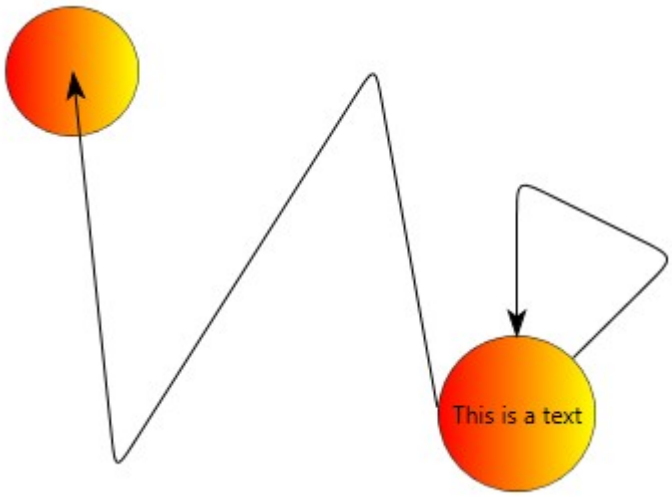
Click for instance on the blue node. This will select it. Click another time. An edit box is displayed inside the node, allowing you to enter a text.



4.2.9 Adjust the link origin and destination points

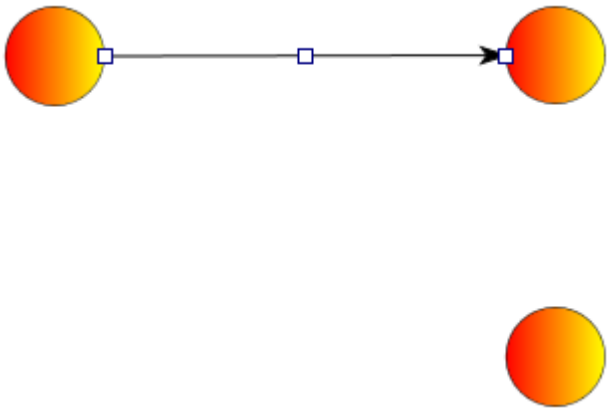
By default, you cannot adjust the extremities of the links. For instance, if you select a link and bring the mouse cursor into the last (or the first) handle of this link, press the left button, move the mouse in another place then relinquish the mouse button, the link springs back again, retrieving its initial position.
However, you can change this behavior by using the **InConnectionMode** property of the destination

node of the link and the **OutConnectionMode** property of the origin node of the link.In such a case, 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.

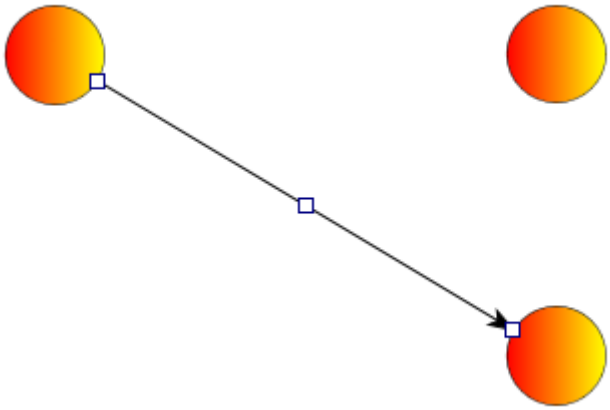


4.2.10 Change the destination or the origin node of a link

You can change interactively the destination or the origin of a link.



You bring the mouse cursor into the third link handle (near the arrow head), press the left button, move the mouse until the isolated node and release the left button. The new destination of the link has changed.



5 Programmatic creation of a diagram

5.1 Overview

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

The main class is the **AddFlow** class that derives from the Control class. It contains a FrameworkElement element that contains the diagram. The **Diagram** property returns this FrameworkElement object.

The Diagram FrameworkElement contains a DrawingVisual object which is used to display the diagram. The **Visual** property returns this DrawingVisual object.

An AddFlow diagram contains two kinds of objects, **Node** objects and **Link** objects. A Link object allows to link 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 Node and Link classes derive from the **Item** class which derives itself from the DrawingVisual class. Therefore, nodes and links are in fact **DrawingVisual** objects.

WARNING: Nodes and links are not controls! We could have decided to choose a design where nodes and links are controls (as we have done for the Silverlight version of AddFlow but in this case, we had not the choice) but this would result in poor speed performance. Using DrawingVisual is the mots lightweight way to perform drawing. It is the best choice if you need to create big diagrams (several thousands of nodes and links).

Notice also that a DrawingVisual object can contains a collection of DrawingVisual objects. We use this feature in the **CauseEffect** example of the DemoFlow sample.

You can obtain the collection of the DrawingVisual objects by using the **Children** property of the Visual object. The Children Collection includes all the nodes, all the links and also all other DrawingVisual objects that could be included in the Visual object of AddFlow.

This Children property is important since it is the only way to obtain the list nodes and links of AddFlow. The good new is that you can use the new **LINQ** technology to select the set of objects you wish. For instance,

if you wish to select all node in an IEnumerable object:

```
IEnumerable<Node> nodes = addflow.Diagram.Visual.Children.OfType<Node>();
```

then, if for instance, you wish to draw them with a red color:

```
foreach (Node node in nodes)
       node.Fill = Brushes.Red;
```

or if you prefer selecting all the nodes in an array:

```
var nodes = addflow.Diagram.Visual.Children.OfType<Node>().ToArray();
foreach (Node node in nodes)
       node.Fill = Brushes.Red;
```

Same thing for the links:

```
IEnumerable<Link> links = addflow.Diagram.Visual.Children.OfType<Link>();
foreach (Link link in links)
       link.Stroke = Brushes.Red;
```

If you wish selecting all the nodes that are selectable:

```
var nodes = addflow.Diagram.Visual.Children.OfType<Node>().Where(
                                                  node => node.IsSelectable == true);
foreach (Node node in nodes)
       node.Fill = Brushes.Red;
```

or if you wish selecting all the links that are stretchable:

```
var links = addflow.Diagram.Visual.Children.OfType<Link>().Where(
                                                  link => link.IsStretchable == true);
foreach (Link link in links)
       link.Stroke = Brushes.Red;
```

5.2 Diagram creation

5.2.1 Our first program

Using Visual Studio .NET, select the menu item **File | New | Project...**. In the Add New Project dialog box, select the **Visual C#** project type, then choose **WPF Application,** change the project name to **AddFlowWPFTutorial1** and click **ok**.

Right-click in the References item underneath the project name and select Add Reference from the context menu. Then, use the **Browse...** button to search for the Lassalle.WPF.Flow.dll and select it.

The initial XAML description of the main application window **Window1.xaml** is the following:

```
<Window x:Class="AddFlowWpfTutorial1.Window1"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="Window1" Height="300" Width="300">
    <Grid>

    </Grid>
</Window>
```

Let us replace it by the following XAML code:

```
<Window x:Class="AddFlowWpfTutorial1.Window1"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:af="clr-namespace:Lassalle.WPF.Flow;assembly=Lassalle.WPF.Flow"
    Title="Window1" Height="300" Width="300">
    <Grid>
        <af:AddFlow Name="addflow" />
    </Grid>
</Window>
```

If you compile and run this application, you will see that it allows to create interactively nodes and links. Nodes can be moved and resized. Links can be stretched. You can also add text inside each node.

However, we are going to create a diagram with C# code. Let us open the code-behind file **Window1.xaml.cs**.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Data;
using System.Windows.Documents;
using System.Windows.Input;
using System.Windows.Media;
using System.Windows.Media.Imaging;
using System.Windows.Navigation;
using System.Windows.Shapes;

namespace AddFlowWPFTutorial1
{
    /// <summary>
    /// Interaction logic for Window1.xaml
    /// </summary>
    public partial class Window1 : Window
    {
        public Window1()
        {
            InitializeComponent();
        }
    }
}
```

Let us replace it by the following:

```
using System;
using System.Linq;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Input;
using System.Windows.Media;
using System.Windows.Media.Imaging;
using Lassalle.WPF.Flow;

namespace AddFlowWPFTutorial1
{
    public partial class Window1 : Window
    {
        public Window1()
        {
           InitializeComponent();

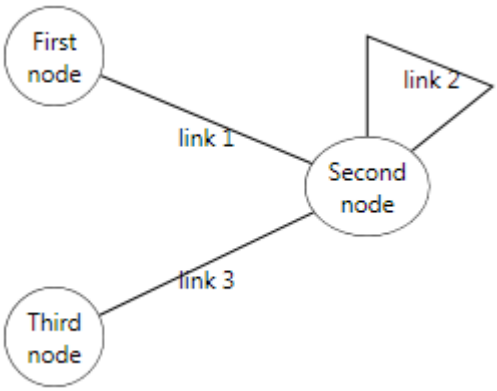
           // Force the addflow templating to be taken into account.
           // If we do not do that, the creation of the diagram will fail
           // because the visual tree of AddFlow would not be created yet.
           this.addflow.ApplyTemplate();
           // ... and create the diagram.
           this.CreateDiagram();
        }

        private void CreateDiagram1()
        {
            // Create 3 nodes
            Node node1 = this.addflow.AddNode(20, 20, 50, 50, "First node");
            Node node2 = this.addflow.AddNode(135, 85, 62, 50, "Second node");
            Node node3 = this.addflow.AddNode(20, 115, 50, 50, "Third node");

            // Create 3 links
            Link link1 = this.addflow.AddLink(node1, node2, "link 1");
            Link link2 = this.addflow.AddLink(node2, node2, "link 2");
            Link link3 = this.addflow.AddLink(node2, node3, "link 3");
        }
    }
```

```
}
```

If we compile and execute this program, it will create 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 not 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 and links are created. First we create the nodes then we create the links. This is because a link cannot exist without its origin and destination nodes.

To create a node, you have to use the **AddNode** method. There is no other method. However, there are 3 versions of this method. In th example above, we use the version that takes 5 parameters and returns the created node. There is also a version of AddNode that takes only the first 4 parameters. For instance, to create the first node, you could have written:

```
Node node1 = this.addflow.AddNode(20, 20, 50, 50);
node1.Text = "First node";
```

There is also a version of AddNode that takes just one parameter of Node type.

```
Node node1 = new Node(20, 20, 50, 50, "First node");
this.addflow.AddNode(node1);
```

To create a link, you have to use the **AddLink** method. There is no other method. However, there are 3 versions of this method. In th example above, we use the version that takes 3 parameters and returns the created link. There is a version of AddLink that takes only the 2 first parameters. For instance, to create the first link, you could have written:

```
Link link1 = this.addflow.AddLink(node1, node2);
link1.Text = "link 1";
```

There is also a version of AddLink that takes just one parameter of Link type.

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

Notice that we are not forced to memorize each node reference. Using LINQ, we could rewrite CreateDiagram1 as follows:

```
private void CreateDiagram1()
{
    // Create 3 nodes
    this.addflow.AddNode(20, 20, 50, 50, "First node");
    this.addflow.AddNode(135, 85, 62.5, 50, "Second node");
```

```
    this.addflow.AddNode(20, 115, 50, 50, "Third node");

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

    // Create 3 links
    this.addflow.AddLink(nodes[0], nodes[1], "link 1");
    this.addflow.AddLink(nodes[1], nodes[1], "link 2");
    this.addflow.AddLink(nodes[1], nodes[2], "link 3");
}
```

In the DemoFlow sample provided with AddFlow, we generally use this method.

Notice that you could also write it using the “one parameter versions of AddNode and Addlink”:

```
 private void CreateDiagram11()
 {
     // Create 3 nodes
     Node node1 = new Node(20, 20, 50, 50, "First node");
     Node node2 = new Node(230, 85, 70, 50, "Second node");
     Node node3 = new Node(20, 160, 50, 50, "Third node");
     this.addflow.AddNode(node1);
     this.addflow.AddNode(node2);
     this.addflow.AddNode(node3);

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

or, using a more compact style:

```
private void CreateDiagram12()
 {
     // Create 3 nodes
     this.addflow.AddNode(new Node(20, 20, 50, 50, "First node"));
     this.addflow.AddNode(new Node(230, 85, 70, 50, "Second node"));
     this.addflow.AddNode(new Node(20, 160, 50, 50, "Third node"));

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

     // Create 3 links
     this.addflow.AddLink(new Link(nodes[0], nodes[1], "link 1"));
     this.addflow.AddLink(new Link(nodes[1], nodes[1], "link 2"));
     this.addflow.AddLink(new Link(nodes[1], nodes[2], "link 3"));
 }
```

5.2.2 Node Properties

The node properties are described in the **Lassalle.WPF.Flow** namespace in the AddFlow help file. Some are inherited from the Item class which inherits from the DrawingVisual class.

We can group them in 4 categories:

- Layout properties: **Location**, **Size**
- Appearance properties: **Text, Tooltip, Geometry, Fill, Stroke, StrokeThickness, DashStyle, TextPosition, TextMargin, Image, ImagePosition, ImageMargin, TextTrimming**

- Graph properties: The **Links** collection property that allow getting all the links (in and out) of the node.

- Behaviour properties: **IsSelected, IsSelectable, IsXMoveable, IsYMoveable, IsXSizeable, IsYSizeable, IsEditable, IsEditing**

- Connection properties defining how a link is attached to a node: **InConnectionMode**, **OutConnectionMode** and **Connector**

5.2.3 Link Properties

The link properties are described in the **Lassalle.WPF.Flow** namespace in the AddFlow help file. Some are inherited from the Item class which inherits from the DrawingVisual class.

We can group them in 3 categories:

- Appearance properties: **Text, Tooltip, LineStyle, ArrowGeometryDst, ArrowGeometryOrg, ArrowGeometryMid, TextPosition, RoundedCornerSize**

- Graph properties: the **Org** property returns/sets the reference of the origin node of the link whereas the **Dst** property returns/sets the reference of the destination node of the link.

- Behaviour properties: IsSelected, IsSelectable, IsStretchable

WARNING: There is also the **Points** collection. A link is composed of seneral segments defined by a collection of points. However, you should not use this collection directly except for serialization purposes. To manipulate the collection the link points, you should use instead the methods **AddPoint, RemovePoint, ClearPoints, SetPoint, GetPoint** and **CountPoints.**

5.2.4 Changing property values

Now let us replace the CreateDiagram1 method by the following CreateDiagram2 method:

```
private void CreateDiagram2()
{
    Node node = null;
    Link link = null;

    // Create 3 yellow nodes.
    // The second node is rectangular
    // and the third one has a Document shape style.

    node = this.addflow.AddNode(20, 20, 50, 50, "First node");
    node.Fill = Brushes.LightYellow;

    node = this.addflow.AddNode(170, 85, 62.5, 50, "Second node");
    node.Fill = Brushes.LightYellow;
    node.Geometry = new RectangleGeometry(new Rect(0, 0, 64, 64));

    node = this.addflow.AddNode(20, 160, 50, 50, "Third node");
    node.Fill = Brushes.LightYellow;
    node.Geometry = Geometry.Parse("M 0,0 H 60 V 40 C 30,30 30,50 0,40 Z");

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

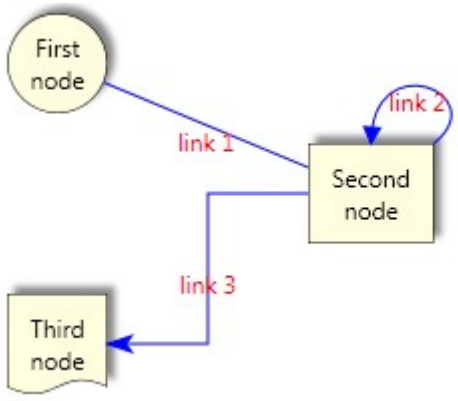
    // Create 3 links.
    // Each link is blue and its ext color is red
```

```
        // The first link has no arrow.
        // The second link has a Bezier style
        // The third link has a "HVH" style.
        link = this.addflow.AddLink(nodes[0], nodes[1], "link 1");
        link.Stroke = Brushes.Blue;
        link.Foreground = Brushes.Red;

        link = this.addflow.AddLink(nodes[1], nodes[1], "link 2");
        link.Stroke = Brushes.Blue;
        link.LineStyle = LineStyle.Bezier;
        link.Foreground = Brushes.Red;
        link.ArrowGeometryDst = Geometry.Parse("M0,0 8,4 0,8 Z");

        link = this.addflow.AddLink(nodes[1], nodes[2], "link 3");
        link.Stroke = Brushes.Blue;
        link.Foreground = Brushes.Red;
        link.LineStyle = LineStyle.HVH;
        link.ArrowGeometryDst = Geometry.Parse("M3,4 L0,0 12,4 0,8 z");
    }
```

If we compile and execute this program, you will see that now, our nodes have distinct shapes. Their fill color is LightYellow. And our links are blue. The reflexive link is a curved line, its text is red. The first link has no arrow.



Notice however that to specify the color of each node, we had to do it for each node, even if the color 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. This is the subject of the following paragraph.

TIP: In the previous examples, to define a node geometry or a link arrow geometry, we have used the PathGeometry Markup Syntax. For instance:

```
node.Geometry = Geometry.Parse("M 0,0 H 60 V 40 C 30,30 30,50 0,40 Z");
```

However, this method, although elegant, has also its drawbacks. We will see later another method that uses the PredefinedGeometry class implemented in the Lassalle.WPF.Geometries assembly.

5.2.5 Default property values

Now let us replace the CreateDiagram2 method by the following CreateDiagram3 method:

```
private void CreateDiagram3()
{
    Node node = null;
    Link link = null;

    // Assign default property values for nodes and links
    this.addflow.NodeModel.Fill = Brushes.LightYellow;
    this.addflow.LinkModel.Stroke = Brushes.Blue;
    this.addflow.LinkModel.Foreground = Brushes.Red;

    // Create 3 yellow nodes.
    // The second node is rectangular
    // and the third one has a Document shape style.

    node = this.addflow.AddNode(20, 20, 50, 50, "First node");

    node = this.addflow.AddNode(170, 85, 62.5, 50, "Second node");
    node.Geometry = new RectangleGeometry(new Rect(0, 0, 64, 64));

    node = this.addflow.AddNode(20, 160, 50, 50, "Third node");
    node.Geometry = Geometry.Parse("M 0,0 H 60 V 40 C 30,30 30,50 0,40 Z");

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

    // Create 3 links.
    // The first link has no arrow.
    // The second link has a Bezier style
    // The third link has a "HVH" style.

    link = this.addflow.AddLink(nodes[0], nodes[1], "link 1");

    link = this.addflow.AddLink(nodes[1], nodes[1], "link 2");
    link.LineStyle = LineStyle.Bezier;
    link.ArrowGeometryDst = Geometry.Parse("M0,0 8,4 0,8 Z");

    link = this.addflow.AddLink(nodes[1], nodes[2], "link 3");
    link.LineStyle = LineStyle.HVH;
    link.ArrowGeometryDst = Geometry.Parse("M3,4 L0,0 12,4 0,8 z");
}
```

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** and the **LinkModel** properties of AddFlow. The type of NodeModel is Node whereas the type of LinkModel is Link. However these objects are not part of AddFlow diagram. Both properties just allow specifying default property values for nodes and links.

For instance, writing:

```
addflow.NodeModel.Fill = Brushes.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** and the **LinkModel** properties have also an interactive effect. Not only the nodes created programmatically will be filled with a LightYellow color but also the nodes created interactively with the mouse. This may be interesting or not, depending on what you intend to do.

5.2.6 Stretching the links

We would like to add segments to our links. The following CreateDiagram4 method demonstrates how to do that.

```
 private void CreateDiagram4()
{
    Node node = null;
    Link link = null;

    // Assign default property values for nodes and links
    this.addflow.NodeModel.Fill = Brushes.LightYellow;
    this.addflow.LinkModel.Stroke = Brushes.Blue;
    this.addflow.LinkModel.Foreground = Brushes.Red;

    // Create 3 yellow nodes.
    // The second node is rectangular
    // and the third one has a Document shape style.

    node = this.addflow.AddNode(20, 20, 50, 50, "First node");

    node = this.addflow.AddNode(170, 85, 62.5, 50, "Second node");
    node.Geometry = new RectangleGeometry(new Rect(0, 0, 64, 64));

    node = this.addflow.AddNode(20, 160, 50, 50, "Third node");
    node.Geometry = Geometry.Parse("M 0,0 H 60 V 40 C 30,30 30,50 0,40 Z");

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

    // Create 3 links.
    // The first link has no arrow.
    // The second link has a Bezier style, the color of its text is red
    // The third link has a "HVH" style.

    link = this.addflow.AddLink(nodes[0], nodes[1], "link 1");

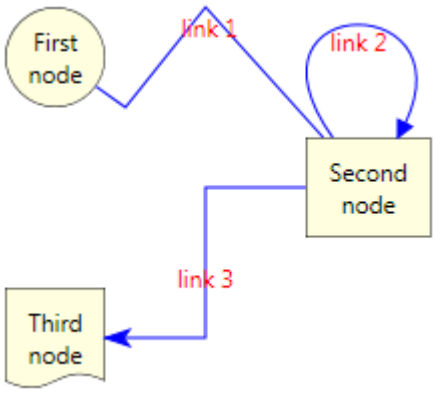
    // Add 2 points (therefore 2 segments) to this first link
    link.AddPoint(new Point(80, 70));
    link.AddPoint(new Point(120, 20));

    link = this.addflow.AddLink(nodes[1], nodes[1], "link 2");
    link.LineStyle = LineStyle.Bezier;
    link.ArrowGeometryDst = Geometry.Parse("M0,0 8,4 0,8 Z");

    // Stretch this reflexive link
    link.SetPoint(new Point(130, 10), 1);
    link.SetPoint(new Point(260, 10), 2);

    link = this.addflow.AddLink(nodes[1], nodes[2], "link 3");
    link.LineStyle = LineStyle.HVH;
    link.ArrowGeometryDst = Geometry.Parse("M3,4 L0,0 12,4 0,8 z");
}
```

If we compile and execute this program, you will see that first link has now 3 segments and the reflexive link is bigger.



To add segments to a link or to alter its shape, you have to use the **AddPoint** method collection of the link (do not use the Points collection!)

You can add points (and therefore segments) to the link 1 because its link line style is Polyline. You could also do that it its link line style was Spline. However, for the other cases (for instance Bezier as for the link 2), you cannot add points. You can however still modify the position of the points, using the **SetPoint** method.The rules for managing the link collection of points are the following:

- 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.
- 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 can change the first point of the Points collection only if the **OutConnectionMode** property of the origin node is NOT set to **ConnectionMode.Center**.
- You can change the last point of the Points collection only if the **InConnectionMode** property of the destination node is NOT set to **ConnectionMode.Center**.
- You can change each other point of the Points collection in any case.

5.3 How to define the node geometry

In the previous examples, to define a node geometry or a link arrow geometry, we have used the PathGeometry Markup Syntax. For instance:

```
node.Geometry = Geometry.Parse("M 0,0 H 60 V 40 C 30,30 30,50 0,40 Z");
```

or

```
link.ArrowGeometryDst = Geometry.Parse("M0,0 8,4 0,8 Z");
```

This is a good and elegant method except if you wish to export the diagram in XAML (it just does not work). However, you can write your own code to define a geometry or better, you can also use the **PredefinedGeometry** class (defined in the Lassalle.Flow.Geometries assembly). In such a case, you could write:

```
node.Geometry = PredefinedGeometry.GetNodeGeometry(NodeShapeStyle.Document);
```

or

```
link.ArrowGeometryDst =
      PredefinedGeometry.GetLinkArrowGeometry(LinkArrowStyle.Arrow, 8, 8);
```

The second and third parameter allows defining the size of the arrow.

If you use this method, you will not face any problem to export the diagram to XAML, as described later in this tutorial.

5.4 Displaying the node content

We have seen that our Node class do not derive from the Control class or the ContentControl class. It derives from the DrawingVisual class. Inside a node, you can place a text, an image or both. As a node is a DrawingVisual object, it can also contain a collection of DrawingVisual objects. It is possible use this feature to customize the node content as demonstrated in the **CauseEffect** example of the DemoFlow sample.

5.4.1 Node text

You can display a label inside a node, using the Text property:

```
node1.Text = "First node";
```

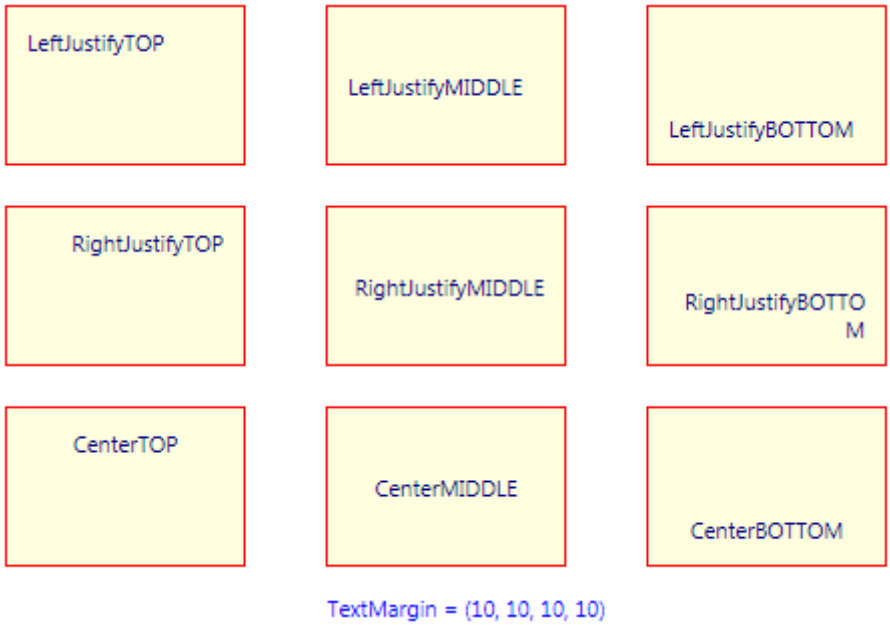
You can also define this label directly when creating the node, as we have seen previously:

```
Node node1 = new Node(20, 20, 50, 50, "First node");
this.addflow.AddNode(node1);
```

or

```
Node node1 = this.addflow.AddNode(20, 20, 50, 50, "First node");
```

The **TextPosition** and **TextMargin** properties allow determining where the image is placed inside the node.



5.4.2 Node image

You can associate an image to a node with the Image property. The **WorkFlow** example in the DemoFlow sample shows how to that. It first creates 3 Image objects.

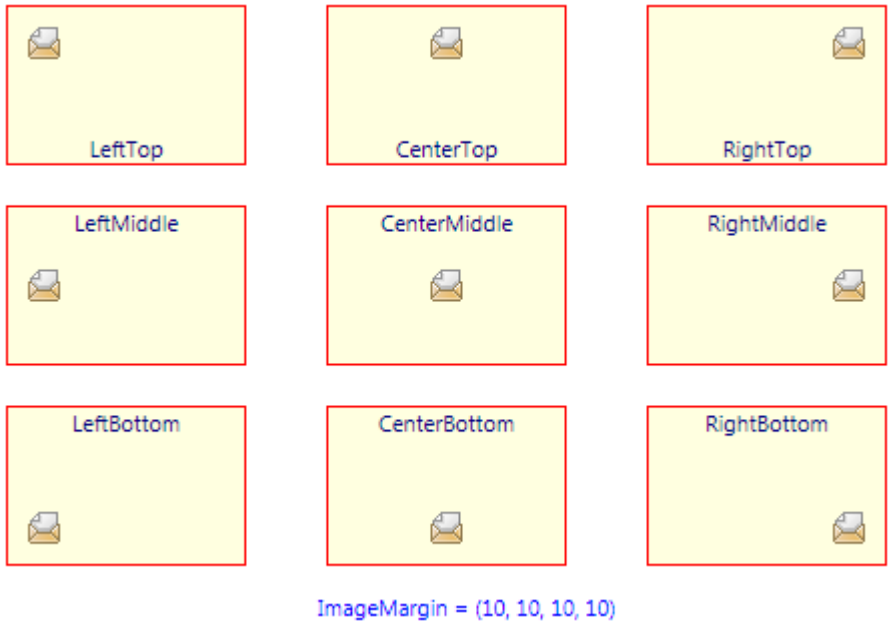
```
Image imageMail = new Image();
imageMail.Source = new BitmapImage(
                        new Uri("pack://application:,,,/Images/mail.png"));
```

```
Image imagePhone = new Image();
imagePhone.Source = new BitmapImage(
                        new Uri("pack://application:,,,/Images/phone.png"));
Image imageDisc = new Image();
imageDisc.Source = new BitmapImage(
                        new Uri("pack://application:,,,/Images/disc.png"));
```

Then, to assign an image to a node, you have just to write:

```
node.Image = imagePhone.Source;
```

The **ImagePosition** and **ImageMargin** properties allow determining where the image is placed inside the node.



5.5 Displaying a shadow for a node or a link

The **FlowChart** example in the DemoFlow sample shows how to that. We use a DropShadowEffect object which is hardware rendered (and therefore quickly rendered):

```
private DropShadowEffect dse;
```

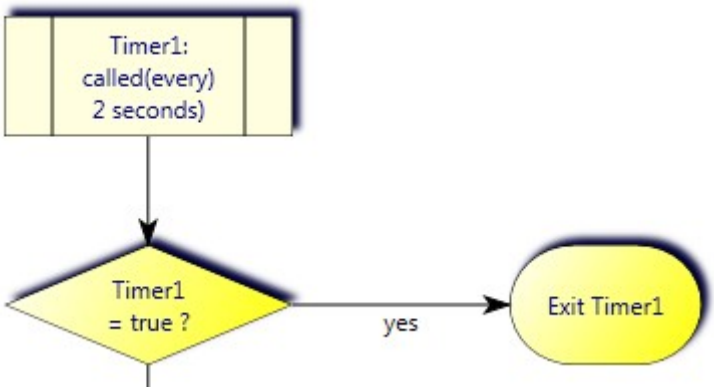
In the constructor of the class, we define it:

```
this.dse = new DropShadowEffect();
this.dse.Color = Colors.Navy;
this.dse.Direction = 45;
```

Then, to display a node shadow, you have just to write:

```
node1.Effect = this.dse;
```

The following figure demonstrates the use of this shadow effect.



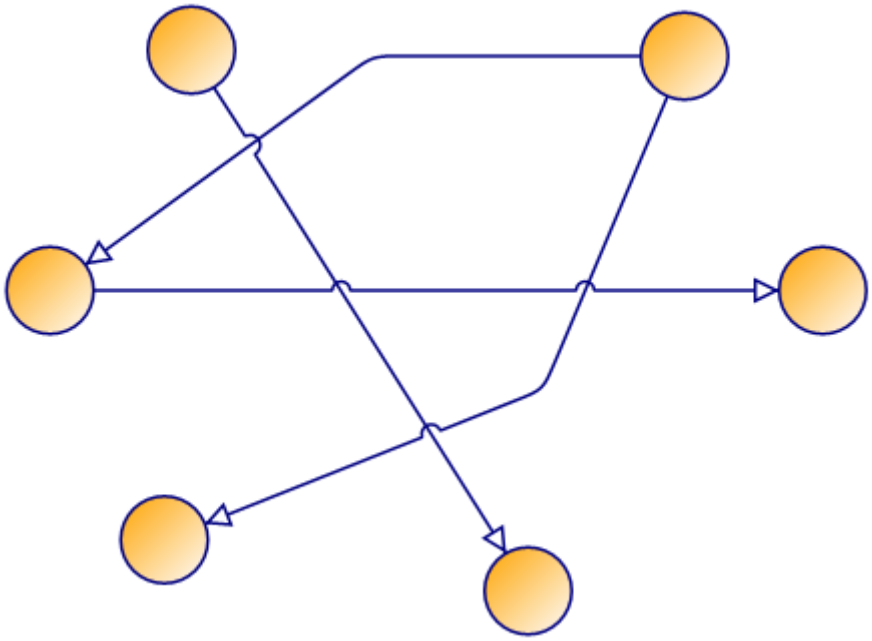
5.6 Displaying link intersections

AddFlow uses a poweful algorithm to find intersections between the link segments.

If the **CanShowJumps** property of the AddFlow control is true then jumps will be displayed at the intersection of this link with other links.

However, this will not work if the link is a curved link (Bezier or Spline).

The size of jumps is determined by the value of the **JumpSize** property of the link. If this value is 0, then no jump will be displayed.



5.7 Selection of items

5.7.1 Interactive selection

You can select a node or a link interactively by clicking it with the mouse.

You can also select several items interactively with the mouse if multi-selection is allowed (in such a case, the **CanMultiSelect** property must be true).

You can select several items 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 or MouseSelection.Selection2. 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 partly 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.

FAQ: 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.

5.7.2 Programmatic selection

You can select a node or a link programmatically with the **IsSelected** property of the Item object:

```
node1.IsSelected = true;
node2.IsSelected = true;
link1.IsSelected = true;
```

5.7.3 SelectedItems collection

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

```
// Make each selected item (node or link) red
foreach (Item item in addflow.SelectedItems)
    item.Stroke = Brushes.Red;
```

WARNING: there is a difference with previous versions (the ActiveX version or the Windows Form version). In the WPF 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.

However, you can implement a “current item” feature. You have just to emphasize the first item of the SelectedItems collection. This is demonstrated in the **State diagram** example of the DemoFlow sample.

5.7.4 Selection events

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.

5.8 Diagram navigation

AddFlow provides four and only four properties to navigate in a diagram (“Network traversals”). Notice that these properties described here are demonstrated in the “**Navig**”diagrams in the **DemoFlow** sample provided with AddFlow.

We have already spoken of these properties.

- **Children**. It is the collection of all the DrawingVisual objects contained in the diagram. Therefore it includes all nodes and links (since nodes and links are DrawingVisual objects) but also all other DrawingVisual objects you may add in the diagram. See the paragraph 4.1 to see how to retrieve a list of nodes or links using LINQ.
- **Links**. It is the collection of links coming to or leaving a node.
- **Org**. It is the origin node of a link.
- **Dst**. It is the destination node of a link.

5.9 Zooming

The Zoom property allows zooming a diagram. It is a double value representing the zoom factor. Its default value is equal to 1. Notice that the zoom is isotropic ensuring a 1:1 aspect ratio.

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?

You can implement this feature using the **ZoomRectangle** method and the **DescendantBounds** property of the Visual object:

```
Rect rc = new Rect();
rc.Union(this.AddFlow.Diagram.Visual.DescendantBounds);
Rect rect = new Rect(0, 0, rc.Size.Width, rc.Size.Height);
this.AddFlow.ZoomRectangle(rect);
```

5.10Serialization

XMLFlow allows serializing an AddFlow diagram. More precisely, this assembly provides a class named **Serial** that offers methods for saving/loading an AddFlow diagram in a file or copying/pasting a portion of it onto the clipboard.

This component is free and its C# source code is provided with AddFlow.

You may also develop your own method for serializing an AddFlow diagram. In such a case, the source code of XMLFlow may be a starting point for you.

XMLFlow is used in the DemoFlow sample (in the MainWindow.xaml.cs file), in the 5 following static methods:

```
static void LoadFile(AddFlow addflow, string strFileName)
{
    XElement root = XElement.Load(strFileName);
    Serial.LoadXML(addflow, XElement.Load(strFileName));
}

static void SaveFile(AddFlow addflow, string strFileName)
{
    XElement xElement = Serial.SaveXML(addflow);
    xElement.Save(strFileName);
}

static void Cut(AddFlow addflow)
{
    Serial.Cut(addflow);
}

static void Copy(AddFlow addflow)
{
    Serial.Copy(addflow);
}

static void Paste(AddFlow addflow)
{
    Serial.Paste(addflow, false);
}
```

5.11 Printing a diagram

AddFlow itself does not offer any printing feature. See the **DiagramPaginator.cs** file of the DemoFlow project to see how you can print a diagram.

Notice that currently, print previewing is not still implemented.

5.12 Exporting a diagram in XAML

AddFlow itself does not offer any exporting feature. However, the file **ExportXaml.xaml.cs** of the DemoFlow project shows how to do that.

If you click on the “Export to XAML” button, all the drawings of the AddFlow control are saved in a XAML file, then this XAML file is loaded and displayed as the content in a Label control placed at the right of the AddFlow control.

This is accomplished by the following routine:

```
void ExportToXaml()
{
    // Add all the Addflow drawings to the Children collection of
    // a DrawingGroup object
    DrawingGroup drawingGroup = new DrawingGroup();
    foreach (DrawingVisual dv in this.addflow.Diagram.Visual.Children)
    {
        DrawingGroup dg = dv.Drawing.Clone();
        dg.Append();
        foreach (DrawingVisual child in dv.Children)
            dg.Children.Add(child.Drawing);
        drawingGroup.Children.Add(dg);
    }

    // Wrap the DrawingGroup in an image
    Image image = new Image();
    image.Stretch = Stretch.None;
    DrawingImage drawingImage = new DrawingImage();
    drawingImage.Drawing = drawingGroup;
    image.Source = drawingImage;
```

```
    // Save the image in a XAML file
    System.Xml.XmlTextWriter writer =
       new System.Xml.XmlTextWriter(@"ExportedDiagram.xaml", null);
    System.Windows.Markup.XamlWriter.Save(image, writer);
    writer.Close();

    // Load the obtained XAML file and display it in a Label control
    // at the right of the AddFlow control
    System.IO.FileStream fs =
       new System.IO.FileStream(@"ExportedDiagram.xaml", System.IO.FileMode.Open);
    Image img = System.Windows.Markup.XamlReader.Load(fs) as Image;
    this.XAMLView.Content = img;
}
```

WARNING: It will not work if you use the PathGeometry Markup Syntax to define the node geometries or the link arrow geometries. You have to define them programmatically.

For instance, if you use the following code to create a node:

```
 node.Geometry = Geometry.Parse("M 0,20 L 30,0 L 60,20 L 30,40 Z");
```

Instead you have to use the following:

```
node.Geometry = PredefinedGeometry.GetNodeGeometry(NodeShapeStyle.Decision);
```

And you cannot define a link arrow geometry using for instance:

```
link2.ArrowGeometryDst = Geometry.Parse("M3,4 L0,0 12,4 0,8 Z");
```

Instead you have to use the following:

```
link2.ArrowGeometryDst =
    PredefinedGeometry.GetLinkArrowGeometry(LinkArrowStyle.Arrow2, 8, 8);
```

This suppose you have added a reference to Lassalle.WPF.Geometries. But you are not forced to use this dll. You may also define your geometry yourself; For instance, the code to create a losange geometry would be the following:

```
node.Geometry = MakeDecisionGeometry(new Rect(0, 0, 64, 64));
```

The MakeDecisionGeometry method could be the following:

```
static Geometry MakeDecisionGeometry(Rect rc)
{
    PathGeometry pathGeometry = new PathGeometry();
    PathFigure pathFigure = new PathFigure();

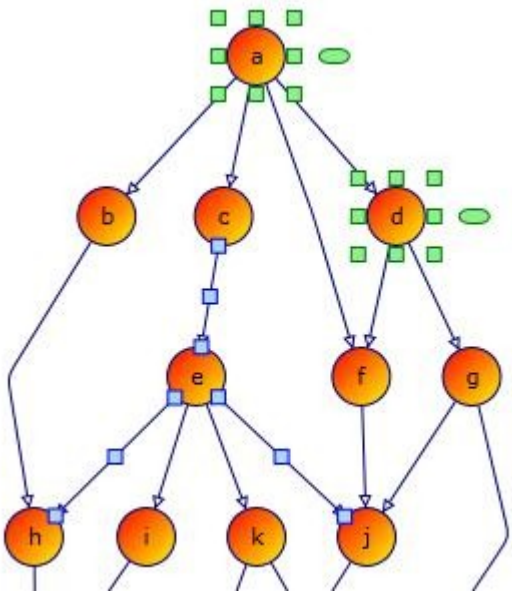
    PolyLineSegment polylineSegment = new PolyLineSegment();
    polylineSegment.Points.Add(new Point(rc.Left, rc.Top + rc.Height / 2));
    polylineSegment.Points.Add(new Point(rc.Left + rc.Width / 2, rc.Top));
    polylineSegment.Points.Add(new Point(rc.Right, rc.Top + rc.Height / 2));
    polylineSegment.Points.Add(new Point(rc.Left + rc.Width / 2, rc.Bottom));
    pathFigure.StartPoint = polylineSegment.Points[0];
    pathFigure.Segments.Add(polylineSegment);
    pathFigure.IsClosed = true;

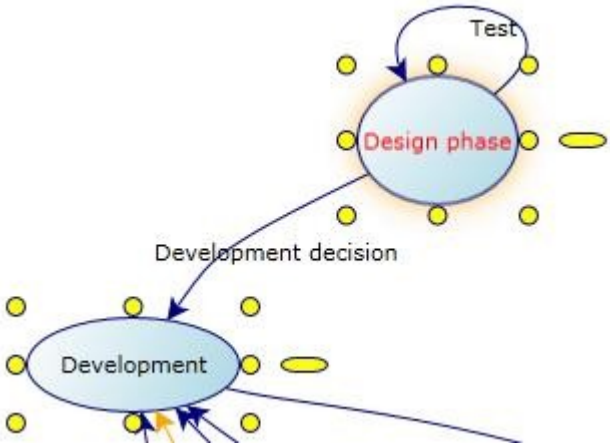
    pathGeometry.Figures.Add(pathFigure);
    return pathGeometry;
}
```

5.13 Customizing the user interface

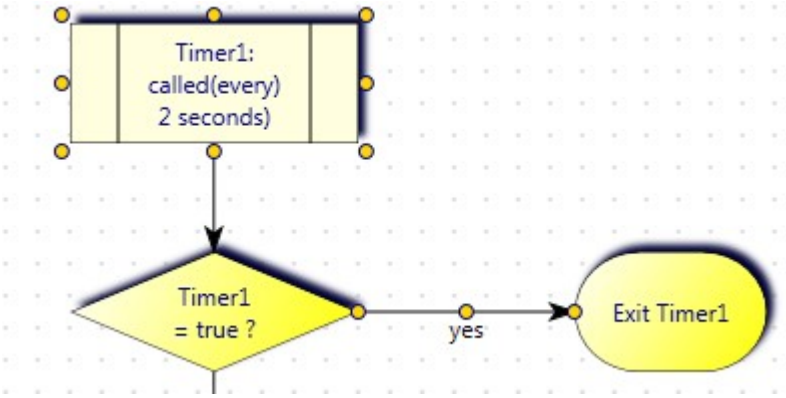
5.13.1 Customizing the handles

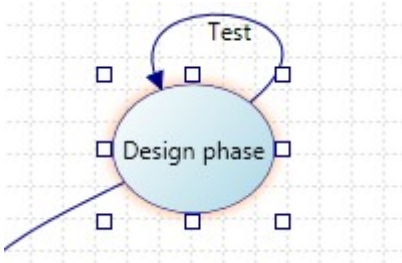
If you look at the different diagram examples provided in the DemoFlow sample, you will see that the thumbs used to resize a node, to rotate a node or to stretch a link are not same.

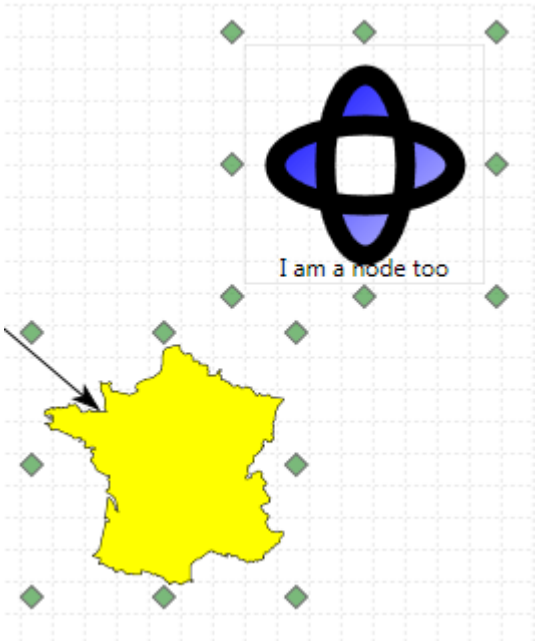


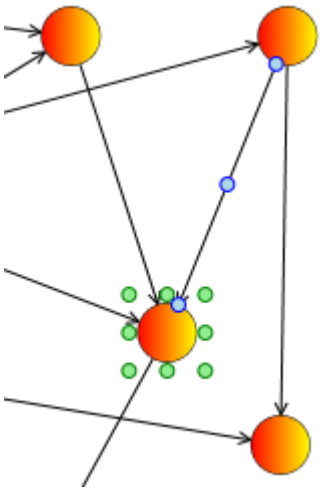












For changing the style of the node resizing handles, you can use the **ResizeHandleStyle** property of AddFlow. For changing the style of the rotation handle, you can use the **RotateHandleStyle** property of AddFlow. And for changing the style of the link stretching handles, you can use the **StretchHandleStyle** property of AddFlow. The type of these properties is **Style**. And you can define a style in xaml.

For instance the following style is defined in the file Flowchart.xaml:

```
<Style TargetType="Thumb" x:Key="handleStyle">
    <Setter Property="Width" Value="8"/>
    <Setter Property="Height" Value="8"/>
    <Setter Property="Cursor" Value="Cross"/>
    <Setter Property="Template">
        <Setter.Value>
          <ControlTemplate TargetType="Thumb">
           <Ellipse x:Name="thumbArea" Stretch="Fill" Fill="Silver" Stroke="Navy"/>
           <ControlTemplate.Triggers>
                 <Trigger Property="IsEnabled" Value="True">
                    <Setter TargetName="thumbArea" Property="Fill" Value="Gold" />
```

```
                </Trigger>
            </ControlTemplate.Triggers>
          </ControlTemplate>
        </Setter.Value>
    </Setter>
</Style>
```

And the following one is used in the file CustomDrawing.xaml:

```
<Style TargetType="Thumb" x:Key="nodeHandleStyle">
    <Setter Property="Width" Value="12"/>
    <Setter Property="Height" Value="12"/>
    <Setter Property="Cursor" Value="Cross"/>
    <Setter Property="Template">
        <Setter.Value>
            <ControlTemplate TargetType="Thumb">
                <Path x:Name="thumbArea" Fill="Silver" Stretch="Fill"
                             Stroke="Black"
                             Opacity="0.6"
                             Data="M 0 6 L 6 0 L 12 6 L 6 12 Z" />
                <ControlTemplate.Triggers>
                   <Trigger Property="IsEnabled" Value="True">
                        <Setter TargetName="thumbArea"
                                Property="Fill" Value="ForestGreen" />
                   </Trigger>
                </ControlTemplate.Triggers>
            </ControlTemplate>
        </Setter.Value>
    </Setter>
</Style>
```

Then, to use these styles, you have just to assign them to the **ResizeHandleStyle**, **RotateHandleStyle** and **StretchHandleStyle** properties of the AddFlow control. For instance:

```
<af:AddFlow Name="addflow"
       GridDraw="True" GridSnap="True" GridColor="DarkGray"
       BorderThickness="6"
       ResizeHandleStyle="{StaticResource nodeHandleStyle}"
       StretchHandleStyle="{StaticResource linkHandleStyle}"/>
```

5.13.2 Customizing the connection of links to a node

5.13.2.1 Generalities

The way a link is connected to a node is managed by this node. Each node has 3 properties for this purpose: **InConnectionMode**, **OutConnectionMode** and **ConnectorStyle**.

The type of the InConnectionMode and OutConnectionMode properties is **ConnectionMode**. It is an enumeration that defines how the link is connected to the node.

Member	**Comment**
Center	Default. The link is directed towards the centre of the node.
Anywhere	The link last (or first) point can be placed anywhere.
Pin	A pin a must be use to connect a link to the node.

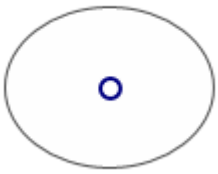
When the mouse is over a node, a **Connector** object is created for that node. This object manages the set of the pins that can be used to connect links. By default, the node connector has only one pin placed at the center of the node. However, as explained later in this paragraph, you may customize a connector so that it contains several pins placed anywhere in the node.

The Pin class exposes the 2 following properties:

Property	Type	Description
In	bool	Determines if the pin accepts "in" links
Out	bool	Determines if the pin accepts "out" links

5.13.2.2 Customizing pins

When the mouse is over a node, a pin (connector) is displayed at the center of the node. It allows creating a link leaving this node towards another node. By default this pin is a circle but you can customize it.



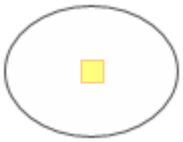
For changing the style of the pin, you can use the **PinStyle** property of AddFlow. The type of this property is **Style**. And you can define a style in xaml.

For instance, the following style is defined in the file StateDiagram.xaml:

```
<Style TargetType="af:Pin" x:Key="pinStyle">
    <Setter Property="Width" Value="12" />
    <Setter Property="Height" Value="12" />
    <Setter Property="Template">
        <Setter.Value>
            <ControlTemplate TargetType="af:Pin">
                <Grid Name="PART_Root" >
                    <Rectangle Stroke="Orange" Fill="Yellow" Opacity="0.5"/>
                </Grid>
            </ControlTemplate>
        </Setter.Value>
    </Setter>
</Style>
```

Then, to use these styles, you have just to assign them to the PinStyle property of the AddFlow control:

```
  <af:AddFlow Name="addflow" GridDraw="True" GridSnap="True"
              SelectionChanged="addflow_SelectionChanged"
              PinStyle="{StaticResource pinStyle}">
    <af:AddFlow.LinkModel>
        <af:Link ArrowGeometryDst="M0,0 8,4 0,8 Z" />
    </af:AddFlow.LinkModel>
</af:AddFlow>
```



Instead of being a circle, it is a yellow rectangle.

5.13.2.3 Customizing the Connector

By default, the node connector contains only one pin placed at the center of the node. However, you can change this default style, using the ConnectorStyle property of AddFlow. This demonstrated in the FlowChart.xaml file where you can find the definition of the following style:

```
<Style TargetType="af:Connector" x:Key="connectorStyle">
```

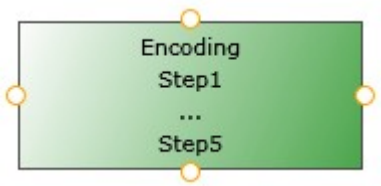
```
<Setter Property="HorizontalContentAlignment" Value="Stretch" />
<Setter Property="VerticalContentAlignment" Value="Stretch" />
<Setter Property="Template">
    <Setter.Value>
        <ControlTemplate TargetType="af:Connector">
            <Grid Margin="-8" Name="PART_Root" >
                <af:Pin x:Name="pin0"
                        VerticalAlignment="Center" HorizontalAlignment="Left" />
                <af:Pin x:Name="pin1"
                        VerticalAlignment="Center" HorizontalAlignment="Right" />
                <af:Pin x:Name="pin2"
                        VerticalAlignment="Top" HorizontalAlignment="Center" />
                <af:Pin x:Name="pin3"
                        VerticalAlignment="Bottom" HorizontalAlignment="Center" />
            </Grid>
        </ControlTemplate>
    </Setter.Value>
</Setter>
</Style>
```

WARNING: The pins must have a name and this name must be “pin0”, “pin1”, “pin2” etc...

Then, to use this style, you have just to assign it to the ConnectorStyle property of the AddFlow control:

```
<af:AddFlow Name="addflow" GridDraw="False" GridSnap="False"
            ResizeHandleStyle="{StaticResource thumbStyle}"
            StretchHandleStyle="{StaticResource thumbStyle}"
            ConnectorStyle="{StaticResource connectorStyle}"
            PinStyle="{StaticResource pinStyle}"
            AllowDrop="True" Drop="addflow_OnDrop">
    <af:AddFlow.NodeModel>
        <af:Node OutConnectionMode="Pin" InConnectionMode="Pin">
            <af:Node.Fill>
                <LinearGradientBrush StartPoint="0,0" EndPoint="1,1">
                    <LinearGradientBrush.GradientStops>
                        <GradientStop Color="White" Offset="0" />
                        <GradientStop Color="Yellow" Offset="1" />
                    </LinearGradientBrush.GradientStops>
                </LinearGradientBrush>
            </af:Node.Fill>
        </af:Node>
    </af:AddFlow.NodeModel>
    <af:AddFlow.LinkModel>
        <af:Link ArrowGeometryDst="M3,4 L0,0 12,4 0,8 z" />
    </af:AddFlow.LinkModel>
</af:AddFlow>
```

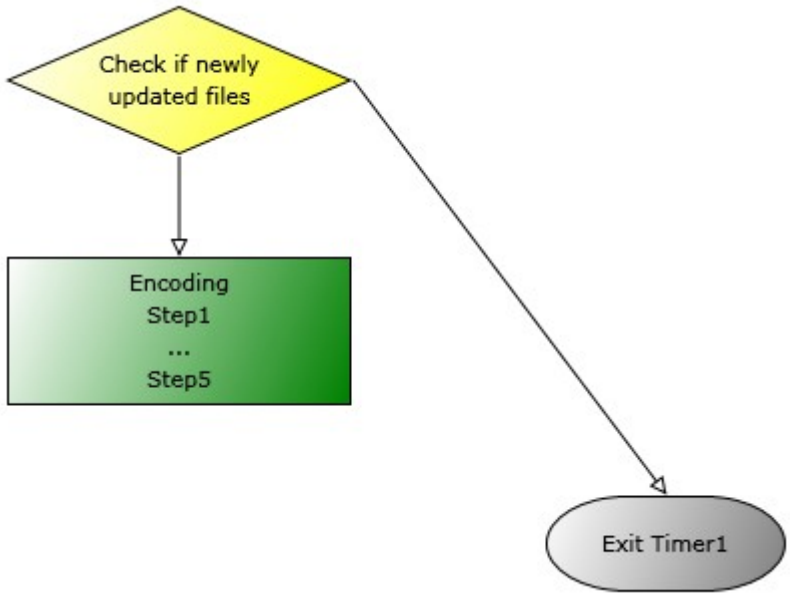
The resulting connector has four pins.



Notice however that the InConnectionMode and OutConnectionMode properties of the node must be set to ConnectionMode.Pin to be able to use those pins to create links. For instance, in the previous case, this is defined for all nodes in the NodeModel property of AddFlow.

```
        <af:Node OutConnectionMode="Pin" InConnectionMode="Pin">
            <af:Node.Fill>
                <LinearGradientBrush StartPoint="0,0" EndPoint="1,1">
```

```
                <LinearGradientBrush.GradientStops>
                    <GradientStop Color="White" Offset="0" />
                    <GradientStop Color="Yellow" Offset="1" />
                </LinearGradientBrush.GradientStops>
            </LinearGradientBrush>
        </af:Node.Fill>
    </af:Node>
```



(Remember that the pins are visible only when the mouse is over the node)

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 **Clear** method.

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 What can be undone and redone?

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

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. And finally, file, print and export operations are clearly not undoable.

WARNING: there is a difference with the previous versions (ActiveX version or the Windows Form version). In the WPF version, only the interactive actions can be undone and redone. In the previous versions, all actions changing a node or a link could be undone or redone. For instance, you could undo a change of the XMoveable property of a node. In the WPF version, you will have to write a custom task to be able to undo a change of the IsXMoveable property. This is explained in the following paragraph.

6.1.5 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. The file **CustomUndo.xaml.cs** of the DemoFlow project shows how to do that: if you double-click on a node, a dialog box allows you changing the text and the tooltip of that node. And then, you can undo this action.

For doing that, we have created a class deriving from the Task class. This new class implements the task consisting in changing both the node text and node tooltip.

```
internal class NodePropertiesTask : Task
{
    private string oldText;
    private string oldTooltip;
    private Node node;

    private NodePropertiesTask() { }

    internal NodePropertiesTask(Node node, string oldText, string oldTooltip)
    {
        this.Code = (Lassalle.WPF.Flow.Action)1002;
        this.node = node;
        this.oldText = oldText;
        this.oldTooltip = oldTooltip;
    }

    public override void Redo()
    {
        this.Undo();
    }

    public override void Undo()
    {
        string oldText = this.node.Text;
        this.node.Text = this.oldText;
        this.oldText = oldText;

        string oldTooltip = this.node.Tooltip as string;
        this.node.Tooltip = this.oldTooltip;
        this.oldTooltip = oldTooltip;
    }
}
```

The code of the mouse double click event handler is the following:

```
private void addflow_MouseDoubleClick(object sender,
                          System.Windows.Input.MouseButtonEventArgs e)
{
    if (this.addflow.SelectedItems != null && this.addflow.SelectedItems.Count > 0)
    {
        if (this.addflow.SelectedItems[0] is Node)
        {
            Node node = this.addflow.SelectedItems[0] as Node;
            NodePropDlg nodePropDlg = new NodePropDlg(node);
            if (nodePropDlg.ShowDialog() == true)
            {
                this.addflow.TaskManager.SubmitTask(new NodePropertiesTask(node,
                                                node.Text, node.Tooltip as string));
                node.Text = nodePropDlg.NodeText;
                node.Tooltip = nodePropDlg.NodeToolTip;
            }
        }
    }
}
```

As you can see, after having closed the dialog box and if it returns true (in such a case, the action is accepted) and before the node receives new values for its Text and Tooltip properties, you can find the following code line:

```
this.addflow.TaskManager.SubmitTask(new NodePropertiesTask(node,
                                node.Text, node.Tooltip as string));
```

This causes the new custom action to be registered in the list of tasks (undo/redo buffer).

6.1.6 Undo/Redo API

The following table gives the list of all properties and methods available to manage the undo/redo feature.

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 Automatic Graph Layout

WARNING: this feature is available without any restriction (nag screen) only if you have purchased a Professional license of AddFlow for WPF.

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 **DemoFlow** sample installed with AddFlow 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.

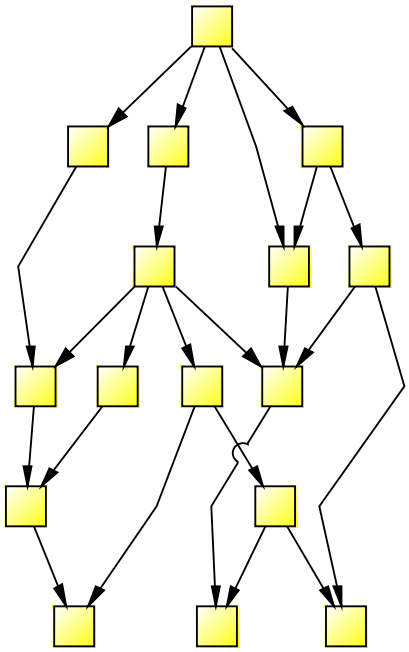
TIP: How to manage so that the layout algorithm applies only to a subset of the graph?

LayoutFlow provides an attached property: **IsLogical**. Only the nodes and links whose IsLogical property is true are involved in each layout. This will allow you to make the layout algorithm to ignore some nodes or links. By default, the IsLogical property is true.

6.2.1 Hierarchic layout

6.2.1.1 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 -

6.2.1.2 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.

6.2.1.3 Limitation

It works with any graph, connected or not.

6.2.1.4 Side Effect

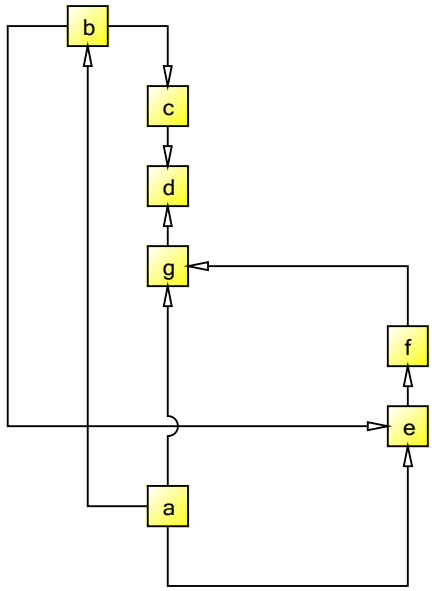
After the layout execution:

- the line style of the links is Polyline
- the InConnectionMode property of the destination node of a link is set to Center.
- the OutConnectionMode property of the origin node of a link is set to Center.

6.2.2 Orthogonal layout

6.2.2.1 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 -

6.2.2.2 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
```

6.2.2.3 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.

6.2.2.4 Side Effect

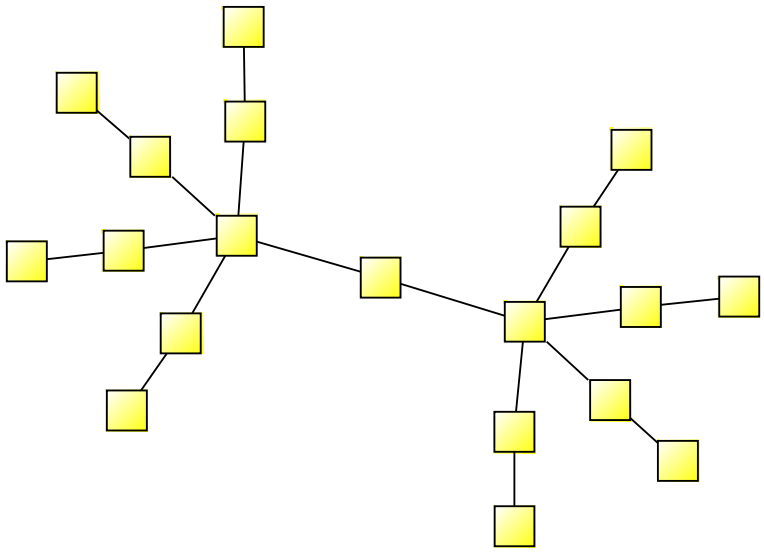
After the layout execution:

- 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 InConnectionMode property of the destination node of a link is set to Anywhere.
- the OutConnectionMode property of the origin node of a link is set to Anywhere.

6.2.3 Symmetric layout

6.2.3.1 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 -

6.2.3.2 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
```

6.2.3.3 Limitation

It works with any graph, connected or not. However, it is recommended to work only with small graphs (less than 200 nodes) because it is using a force-directed method and force-directed methods are using considerable computational resources.

6.2.3.4 Side Effect

After the layout execution:

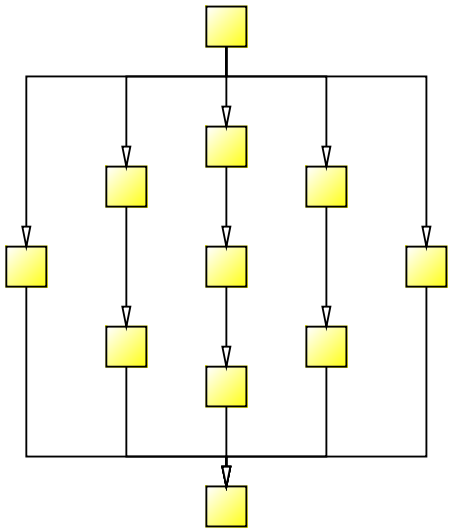
- the line style of the links is Polyline and each link is composed of only one segment.
- the InConnectionMode property of the destination node of a link is set to Center.
- the OutConnectionMode property of the origin node of a link is set to Center.

6.2.4 Series-parallel layout

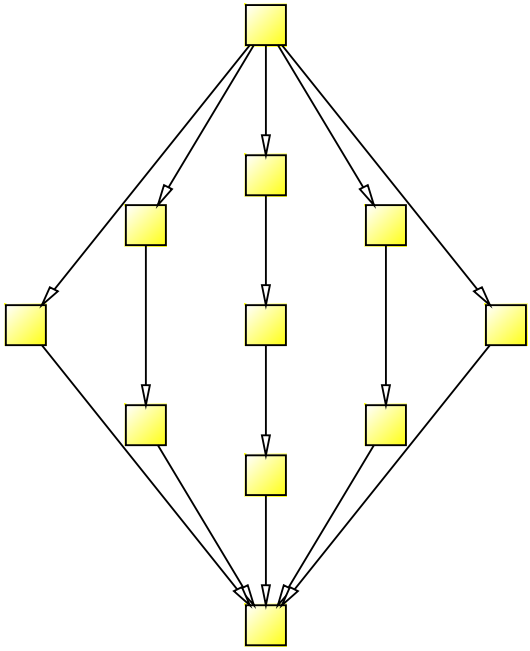
6.2.4.1 Purpose

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.

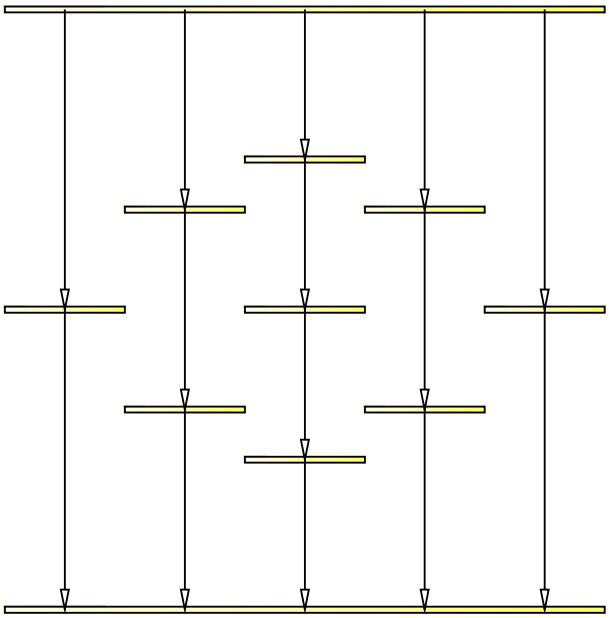
We use an algorithm (described in the book "Drawing Graphs" Michael Kaufmann - Dorothea Wagner) that allows drawing series-parallel digraphs with as much symmetry as possible.



- SP layout: DrawingStyle = BusOrthogonalDrawing



- SP layout: DrawingStyle = StraightLine -



- SP layout: DrawingStyle = VisibilityDrawing -

6.2.4.2 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.

6.2.4.3 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).

6.2.4.4 Side Effect

After the layout execution:

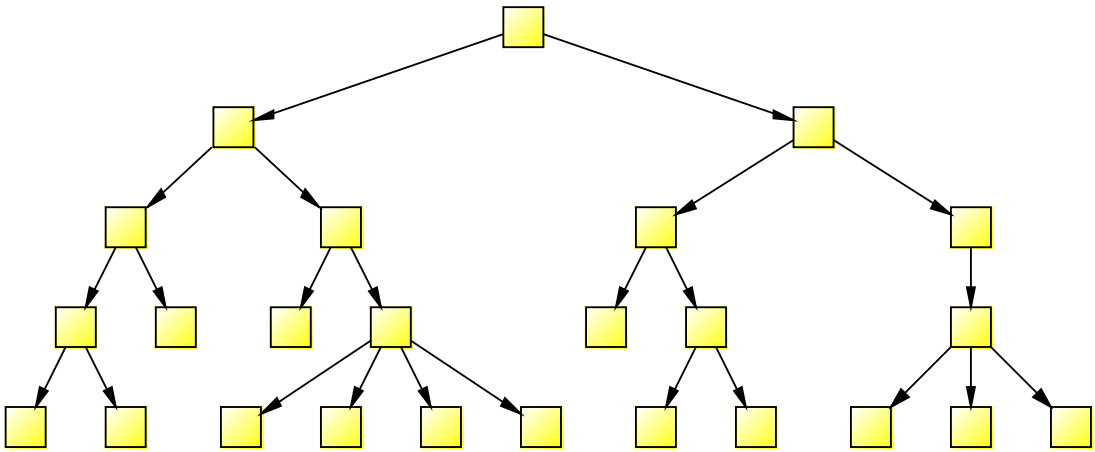
- the line style of the links is Polyline. Moreover, if the DrawingStyle property is not BusOrthogonalDrawing, then each link is composed of only one segment.
- the InConnectionMode property of the destination node of a link is set to Center.
- the OutConnectionMode property of the origin node of a link is set to Center.

6.2.5 Tree layout

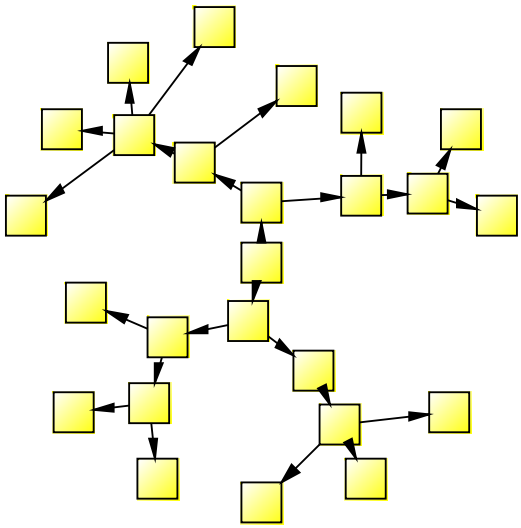
6.2.5.1 Purpose

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 -

6.2.5.2 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.

6.2.5.3 Limitation

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

6.2.5.4 Side Effect

After the layout execution:

- the line style of the links is Polyline
- the InConnectionMode property of the destination node of a link is set to Center.
- the OutConnectionMode property of the origin node of a link is set to Center.

6.3 Data Customization

6.3.1 Framework's Tag property

Nodes and links do not offer a Tag property but it is not a problem because you can use the Framework's Tag dependency property and you can attach it to a node or a link. This is demonstrated in the **CauseEffect** example provided in the DemoFlow sample. For instance, to assocaite the string "Financial" to a node:

```
node.SetValue(FrameworkElement.TagProperty, "Financial");
```

And to retrieve that string:

```
string s = node.ReadLocalValue(FrameworkElement.TagProperty) as string;
```

6.3.2 Attached properties

Attached properties is an Extensibilty mechanism that can be used for storing any data with each node or each link.

This feature is used in the AlgoFlow component whose source code is installed with AddFlow.

For instance, when you write a depth first search algorithmto visit all the nodes of a graph, you need a boolean variable to mark a node as "visited". You can use an attached property for that. You can declare it as follows:

```
  static readonly DependencyProperty IsVisitedProperty =
     DependencyProperty.RegisterAttached("IsVisited", typeof(bool),
             typeof(Connect), new FrameworkPropertyMetadata(false));

 public static bool GetIsVisited(DependencyObject item)
 {
     if (item == null)
         return false;
     return (bool)item.GetValue(IsVisitedProperty);
 }

 public static void SetIsVisited(DependencyObject item, bool value)
 {
     if (item != null)
         item.SetValue(IsVisitedProperty, value);
 }
```

Then you can use it to mark a node as visited:

```
Connect.SetIsVisited(node, true);
```

6.3.3 Derivation of Node and Link classes

Using inheritance, you can create a new class by adding to or otherwise modifying an existing class. You can do that with the Node and Link classes and add custom data to each class.

6.4 Conversion guide from previous versions

6.4.1 Introduction

As you will see, some features are removed, some features are working differently and some features are renamed.

6.4.1.1 Removed Features

- The link jumps (intersection display) feature will be implemented in the next version (with a faster algorithm).
- A Routing feature will also be implemented in the next version.
- The Hidden property is not implemented and the workaround will be to a transparent brush to display the item.

6.4.1.2 Features that work differently

- Node shapes and link arrows are implemented using geometries.
- The Nodes and Items collections disappear because we prefer to use the LINQ technology, as explained at the beginning of the paragraph 4.
- The ReadXml and WriteXml methods (used in the .NET version) disappear because AddFlow does not offer any serialization feature: you have to use XMLFlow instaed.

6.4.1.3 Renamed features

The name of many boolean properties (except the properties that start with the «Can» prefix like CanMoveNode) starts with the «Is» prefix. For instance, the Selected property is renamed IsSelected. This is a trend in the WPF world.

Some events have been also renamed. For instance:

- AfterAddLink is renamed LinkCreated
- BeforeAddlink is renamed LinkCreating

We have done so because it is the wording used in WPF.

6.4.2 Conversion guide from AddFlow ActiveX

6.4.2.1 AddFlow properties

AddFlow ActiveX	**AddFlow for WPF**
AdjustOrg	addflow.NodeModel.OutConnectionMode
AdjustDst	addflow.NodeModel.InConnectionMode
Alignment	addflow.NodeModel.TextPosition
AllowArrowKeys	None
ArrowDst	addflow.LinkModel.ArrowGeometryDst
ArrowMid	addflow.LinkModel.ArrowGeometryMid
ArrowOrg	addflow.LinkModel.ArrowGeometryOrg
Autorouting	None
AutoSize	None

AutoScroll	addflow.CanDragScroll
BackColor	addflow.Background
BackMode	None
BackPicture	None
BorderStyle	None
CanChangeDst	addflow.CanChangeDst
CanChangeOrg	addflow.CanChangeOrg
CanDrawNode	addflow.CanDrawNode
CanDrawLink	addflow.CanDrawLink
CanFireError	addflow.CanFireError
CanMoveNode	addflow.CanMoveNode
CanMultiLink	addflow.CanMultiLink
CanReflexLink	addflow.CanReflexLink
CanSizeNode	addflow.CanSizeNode
CanStretchLink	addflow.CanStretchLink
CanUndoRedo	addflow.CanUndoRedo
CustomShapeIndex	None
CustomShapes	None
DisplayHandles	None
DrawColor	addflow.NodeModel.Stroke addflow.LinkModel.Stroke
DrawStyle	addflow.NodeModel.DashStyle addflow.LinkModel.DashStyle
DrawWidth	addflow.NodeModel.StrokeThickness addflow.LinkModel.StrokeThickness
EditMode	addflow.CanLabelEdit addflow.NodeModel.IsEditable
Ellipsis	addflow.NodeModel.TextTrimming
FillColor	addflow.NodeModel.Fill
Font	addflow.NodeModel.FontFamily, addflow.NodeModel.FontSize, ... addflow.LinkModel. FontFamily, addflow.NodeModel.FontSize, ...
TextColor	addflow.NodeModel.Foreground addflow.LinkModel.Foreground
GridColor	addflow.GridColor
GridStyle	addflow.GridStyle
Hidden	None
JumpSize	None
LastUserAction	None
LinkCreationMode	None
LinkingHandleSize	addflow.StretchThumStyle
LinkStyle	addflow.LinkModel.LineStyle
LogicalOnly	None
MaxDegree	None
MaxInDegree	None
MaxOutDegree	None
MouseIcon	addflow.Cursor
MousePointer	addflow.Cursor
MultiSel	addflow.MultiSel
Nodes	None (use LINQ)
NoPrefix	None
OrientedText	None
OrthogonalDynamic	None
PicturePosition	addflow.NodeModel.ImagePosition
Pictures	None

PointedArea	addflow.PointedArea
PointedLink	addflow.PointedItem
PointedNode	addflow.PointedItem
ProportionalBars	None
ReadOnly	addflow.IsEnabled
RedoCode	addflow.TaskManager.RedoCode
RemovePointAngle	addflow.RemovePointDistance
Repaint	Use the methods BeginUpdate and EndUpdate
Rigid	None
RoundedCorner	None
RoundedCornerSize	None
RouteGrain	None
RouteMinDistance	None
RouteStartMethod	None
ScrollBars	None
ScrollTrack	None
ScrollWheel	None
SelectAction	addflow.MouseSelection
SelectedLink	addflow.SelectedItem
SelectedNode	addflow.SelectedItem
SelectionHandleSize	addflow.ResizeThumbStyle
SelectMode	None
SelLinks	addflow.SelectedItems
SelNodes	addflow.SelectedItems
Shadow	None
ShadowColor	None
Shape	None
ShapeOrientation	None
ShowGrid	addflow.GridDraw
ShowJump	None
ShowPropertyPages	None
ShowToolTip	None
SkipUndo	addflow.TaskManager.SkipUndo
SnapToGrid	addflow.GridSnap
SizeArrowDst	addflow.LinkModel.ArrowGeometryDst
SizeArrowMid	addflow.LinkModel.ArrowGeometryMid
SizeArrowOrg	addflow.LinkModel.ArrowGeometryOrg
StretchingPoint	None
Tag	addflow.Tag
Transparent	addflow.NodeModel.Transparent
UndoCode	addflow.TaskManager.UndoCode
UndoSize	addflow.TaskManager.UndoLimit
XExtent	None
XGrid	addflow.GridSize.Width
XScroll	addflow.ScrollPosition.X
XShadowOffset	None
XZoom	addflow.Zoom
YExtent	None
YGrid	addflow.GridSize.Height
YScroll	addflow.ScrollPosition.Y
YShadowOffset	None
YZoom	addflow.Zoom

6.4.2.2 AddFlow methods

AddFlow ActiveX	AddFlow for WPF
BeginAction	addflow.TaskManager.BeginAction
CanPaste	None
CanRedo	addflow.TaskManager.CanRedo
CanUndo	addflow.TaskManager.CanUndo
Copy	None
DeleteSel	addflow.Delete
DeleteMarked	None
DisplayPropertyPage	None
EndAction	addflow.TaskManager.EndAction
ExportPicture	None
GetLinkAtPoint	GetItemAt
GetNodeAtPoint	GetItemAt
GetVersion	None
IsChanged	addflow.IsChanged
IsSelChanged	None
LoadFile	None (use XMLFlow)
LoadMemory	None (use XMLFlow)
Paste	None
Redo	addflow.TaskManager.Redo
Refresh	None
SaveFile	None (use XMLFlow)
SaveImage	None
SaveMemory	None (use XMLFlow)
SelectAll	Addflow.SelectAll
SelectRectangle	None
SetChangedFlag	addflow.SetChangedFlag
SetSelChangedFlag	None
StartEdit	node.BeginEdit and node.EndEdit
Undo	addflow.TaskManager.Undo
ZoomRectangle	Addflow.ZoomRectangle

6.4.2.3 AddFlow events

AddFlow ActiveX	AddFlow for WPF
AfterAddLink	addflow.LinkCreated
AfterAddNode	addflow.NodeCreated
AfterEdit	addflow.AfterEdit
AfterMove	addflow.SelectedNodesLayouting
AfterResize	addflow.SelectedNodesLayouting
AfterSelect	addflow.SelectionChanged
AfterStretch	addflow.SelectedLinkStretching
BeforeAddLink	addflow.LinkCreating
BeforeAddNode	addflow.NodeCreating
BeforeChangeDst	addflow.DstChanging
BeforeChangeOrg	addflow.OrgChanging
BeforeEdit	addflow.BeforeEdit
Error	Addflow.Error
Scroll	None

6.4.2.4 Node properties

AddFlow ActiveX	AddFlow for WPF
Alignment	node.TextPosition
AutoSize	None
BackMode	None
DrawColor	node.Stroke
DrawStyle	node.DashStyle
DrawWidth	node.StrokeThickness
EditMode	node.IsEditable
FillColor	node.Fill
Font	node.FontFamily, node.FontSize, etc...
ForeColor	node.Foreground
Height	node.Size.Height
Hidden	None
Index	None
InLinks	None
Key	None (see previous paragraph about Data Customization)
Left	node.Location.X
Links	node.Links
Logical	None
Marked	None (see previous paragraph about Data Customization)
MaxDegree	None
MaxInDegree	None
MaxOutDegree	None
Moveable	None
OutLinks	None
Picture	Node.Image
PictureIndex	None
PicturePosition	node.ImagePosition
Selectable	node.IsSelectable
Selected	node.IsSelected
Shadow	None
Shape	node.Geometry
ShapeOrientation	None
Sizable	None
Tag	None (see previous paragraph about Data Customization)
TagVariant	None (see previous paragraph about Data Customization)
Text	node.Text
Tooltip	node.Tooltip
Top	node.Location.Y
Transparent	None
UserData	None (see previous paragraph about Data Customization)
Width	node.Size.Width
xMoveable	node.IsXMoveable
xTextMargin	node.TextMargin
xScrollable	None
xSizeable	node.IsXSizeable
yMoveable	node.IsYMoveable
yTextMargin	node.TextMargin
yScrollable	None
ySizeable	node.IsYSizeable
ZOrder	Use Addflow methods SendToBack, BringToFront, SendBackward,

	BringForward
ZOrderIndex	Use Addflow methods SendToBack, BringToFront, SendBackward, BringForward

6.4.2.5 Node methods

AddFlow ActiveX	**AddFlow for WPF**
Clone	None
EnsureVisible	node.BringIntoView
GetLinkedNode	None
PropertyPage	None

6.4.2.6 Link properties

AddFlow ActiveX	**AddFlow for WPF**
AdjustDst	link.Dst.InConnectionMode
AdjustOrg	link.Org.OutConnectionMode
ArrowDst	link.ArrowGeometryDst
ArrowMid	link.ArrowGeometryMid
ArrowOrg	link.ArrowGeometryOrg
BackMode	None
DrawColor	link.Stroke
DrawStyle	link.DashStyle
DrawWidth	link.StrokeThickness
ExtraPoints	link.Points
Dst	link.Dst
Font	link.Font
ForeColor	link.Foreground
Hidden	None
InIndex	None
Key	None (see previous paragraph about Data Customization)
LinkStyle	link.LineStyle
Logical	None
Marked	None (see previous paragraph about Data Customization)
Org	link.Org
OrientedText	None
OrthogonalDynamic	None
OutIndex	None
Rigid	None
RoundedCorner	link.RoundedCornerSize
Selectable	link.IsSelectable
Selected	link.IsSelected
ShowJump	None
SizeArrowDst	link.ArrowGeometryDst
SizeArrowMid	link.ArrowGeometryMid
SizeArrowOrg	link.ArrowGeometryOrg
Stretchable	link.IsStretchable
Tag	None (see previous paragraph about Data Customization)
TagVariant	None (see previous paragraph about Data Customization)
Text	link.Text
TextSegment	None
Tooltip	link.Tooltip
UserData	None (see previous paragraph about Data Customization)

ZOrder	Use Addflow methods SendToBack, BringToFront, SendBackward, BringForward
ZOrderIndex	Use Addflow methods SendToBack, BringToFront, SendBackward, BringForward

6.4.2.7 Link methods

AddFlow ActiveX	**AddFlow for WPF**
Clone	None
EnsureVisible	link.BringIntoView
PropertyPage	None
Reverse	link.Reverse

6.4.3 Conversion guide from AddFlow for .NET

6.4.3.1 AddFlow properties

AddFlow for .NET	AddFlow for WPF
BackColor	addflow.Background
BorderStyle	None
CanChangeDst	addflow.CanChangeDst
CanChangeOrg	addflow.CanChangeOrg
CanDrawNode	addflow.CanDrawNode
CanDragScroll	addflow.CanDragScroll
CanDrawLink	addflow.CanDrawLink
CanFireError	addflow.CanFireError
CanMoveNode	addflow.CanMoveNode
CanMultiLink	addflow.CanMultiLink
CanReflexLink	addflow.CanReflexLink
CanSizeNode	addflow.CanSizeNode
CanStretchLink	addflow.CanStretchLink
CanUndoRedo	addflow.CanUndoRedo
DisplayHandles	None
Extent	None (use addflow.Diagram.Visual.DescendantBounds.Size)
Grid.Color	addflow.GridColor
Grid.Draw	addflow.GridDraw
Grid.Style	addflow.GridStyle
Grid.Size	Addflow.GridSize
Grid.Snap	Addflow.GridSnap
Items	None (use LINQ)
JumpSize	None
LinkCreationMode	None
LinkingHandleSize	addflow.StretchThumbStyle
MouseAction	addflow.MouseSelection
MultiSel	addflow.CanMultiSelect
Nodes	None (use LINQ)
Images	None
PointedArea	None
PointedItem	None
RedoCode	addflow.TaskManager.RedoCode
RemovePointAngle	addflow.RemovePointDistance
SelectedItem	None
ScrollPosition	None (see ScrollViewer property)
SelectionHandleSize	addflow.ResizeThumbStyle
SelectedItems	addflow.SelectedItems
ShowToolTips	None
SkipUndo	addflow.TaskManager.SkipUndo
StretchingPoint	None
UndoCode	addflow.TaskManager.UndoCode
UndoSize	addflow.TaskManager.UndoLimit
Zoom	addflow.Zoom

6.4.3.2 AddFlow methods

AddFlow for .NET	AddFlow for WPF

AddLink	addflow.AddLink
AddToLastUserAction	addflow.TaskManager.AddToLastUserAction
BeginAction	addflow.TaskManager.BeginAction
BeginUpdate	addflow.BeginUpdate
CanRedo	addflow.TaskManager.CanRedo
CanUndo	addflow.TaskManager.CanUndo
CreateLink	addflow.AddLink
DeleteSel	addflow.Delete
EndAction	addflow.TaskManager.EndAction
EndUpdate	addflow.Update
ExportMetafile	None
GetDiagramSize	None
GetItemAt	addflow.GetItemAt
GetItemsInRectangle	None
IsChanged	addflow.IsChanged
IsSelChanged	None
Redo	addflow.TaskManager.Redo
ReadXML	None (use XMLFlow)
Render	None
ResetUndoRedo	addflow.TaskManager.Clear
SelectAll	addflow.SelectAll
SelectRectangle	None
SetChangedFlag	addflow.SetChangedFlag
SetSelChangedFlag	None
StartEdit	node.BeginEdit and node.EndEdit
SubmitTask	addflow.TaskManager.SubmitTask
Undo	addflow.TaskManager.Undo
WriteXML	None (use XMLFlow)
ZoomPoint	None
ZoomRectangle	addflow.ZoomRectangle

6.4.3.3 AddFlow events

AddFlow for .NET	**AddFlow for WPF**
AfterAddLink	addflow.LinkCreated
AfterAddNode	addflow.NodeCreated
AfterChangeDst	addflow.DstChanged
AfterChangeOrg	addflow.OrgChanged
AfterEdit	addflow.AfterEdit
AfterMove	addflow.SelectedNodesLayouting
AfterRemoveLink	addflow.NodeDeleted
AfterRemoveNode	addflow.LinkDeleted
AfterResize	addflow.SelectedNodesLayouting
AfterSelect	addflow.SelectionChanged
AfterStretch	addflow.SelectedLinkStretching
BeforeAddLink	addflow.LinkCreating
BeforeAddNode	addflow.NodeCreating
BeforeChangeDst	addflow.DstChanging
BeforeChangeOrg	addflow.OrgChanging
BeforeEdit	addflow.BeforeEdit
BeforeReadXMLLink	None (use XMLFlow)

BeforeReadXMLNode	None (use XMLFlow)
BeforeRemoveLink	addflow.NodeDeleting
BeforeRemoveNode	addflow.LinkDeleting
BeforeWriteXMLLink	None (use XMLFlow)
BeforeWriteXMLNode	None (use XMLFlow)
CancelEdit	addflow.CancelEdit
DiagramOwnerDraw	None
Error	addflow.Error
ExtentChange	addflow.ExtentChanged
LinkOwnerDraw	None
NodeOwnerDraw	None
SelectionChange	addflow.SelectionChanged
ReadXMLLinkExtraData	None (use XMLFlow)
ReadXMLNodeExtraData	None (use XMLFlow)
WriteXMLLinkExtraData	None (use XMLFlow)
WriteXMLNodeExtraData	None (use XMLFlow)

6.4.3.4 Node properties

AddFlow for .NET	**AddFlow for WPF**
Alignment	node.TextPosition
AutoSize	None
BackMode	None
DrawColor	node.Stroke
DrawStyle	node.DashStyle
DrawWidth	node.StrokeThickness
EditMode	node.IsEditable
FillColor	node.Fill
Font	node.FontFamily, node.FontSize, etc...
ForeColor	node.Foreground
Height	node.Size.Height
Hidden	None
Index	None
InLinks	None
Left	node.Location.X
Links	node.Links
Logical	None
MaxDegree	None
MaxInDegree	None
MaxOutDegree	None
Moveable	None
OutLinks	None
ImageIndex	node.Image
ImageLocation	Node.ImageMargin
ImagePosition	node.ImagePosition
Selectable	node.IsSelectable
Selected	node.IsSelected
Shadow	None
Shape	node.Geometry
Sizable	None
Tag	None (see previous paragraph about Data Customization)

Text	node.Text
TextMargin	node.TextMargin
Tooltip	node.Tooltip
Top	node.Location.Y
Transparent	None
Width	node.Size.Width
xMoveable	node.IsXMoveable
xSizeable	node.IsXSizeable
yMoveable	node.IsYMoveable
ySizeable	node.IsYSizeable
ZOrder	Use Addflow methods SendToBack, BringToFront, SendBackward, BringForward

6.4.3.5 Node methods

AddFlow for .NET	**AddFlow for WPF**
Clone	None
BringIntoView	node.BringIntoView
GetLinkedNode	None
Remove	

6.4.3.6 Link properties

AddFlow for .NET	**AddFlow for WPF**
AdjustDst	link.Dst.InConnectionMode
AdjustOrg	link.Org.OutConnectionMode
ArrowDst	link.ArrowGeometryDst
ArrowMid	link.ArrowGeometryMid
ArrowOrg	link.ArrowGeometryOrg
BackMode	None
DrawColor	link.Stroke
DrawStyle	link.DashStyle
DrawWidth	link.StrokeThickness
Points	link.Points
Dst	link.Dst
Font	link.Font
TextColor	link.Foreground
Hidden	None
Line.Style	link.LineStyle
Line.OrthogonalDynamic	None
Line.RoundedCorner	link.RoundedCornerSize
Logical	None
Org	link.Org
OrientedText	None
Rigid	None
Selectable	link.IsSelectable
Selected	link.IsSelected
Jump	None (next version)
Stretchable	link.IsStretchable
Tag	None (see previous paragraph about Data Customization)
Text	link.Text
TextSegment	None
Tooltip	link.Tooltip

ZOrder	Use Addflow methods SendToBack, BringToFront, SendBackward, BringForward

6.4.3.7 Link methods

AddFlow for .NET	**AddFlow for WPF**
Clone	None
EnsureVisible	link.BringIntoView
Reverse	link.Reverse