Syncfusion
Deliver innovation with ease

Essential Studio 2013 Volume 4 - v.11.4.0.26

Essential Diagram for Windows Forms

Contents

1 Overview

This section covers information on Essential Diagram control, its key features, and prerequisites to use the control, its compatibility with various OS and browsers and finally the documentation details complimentary with the product. It comprises the following sub sections:

1.1 Introduction To Essential Diagram

Essential Diagram is an extensible and high-performance .NET diagramming framework for Windows Applications. It can be used for developing Microsoft Visio-like interactive graphics and diagramming applications. It stores graphical objects in a node graph and renders those objects onto the screen. Essential Diagram supports both vector and raster graphics on the drawing surface.

Essential Diagram lets users create interactive diagrams easily using the Diagram Builder utility included with the Diagram. It explicitly lays out diagram objects, or allows our built-in layout managers to handle the job, making complex layout diagrams a snap. Using the Symbol Designer utility, you can create domain or business specific symbols by using custom shapes and images.

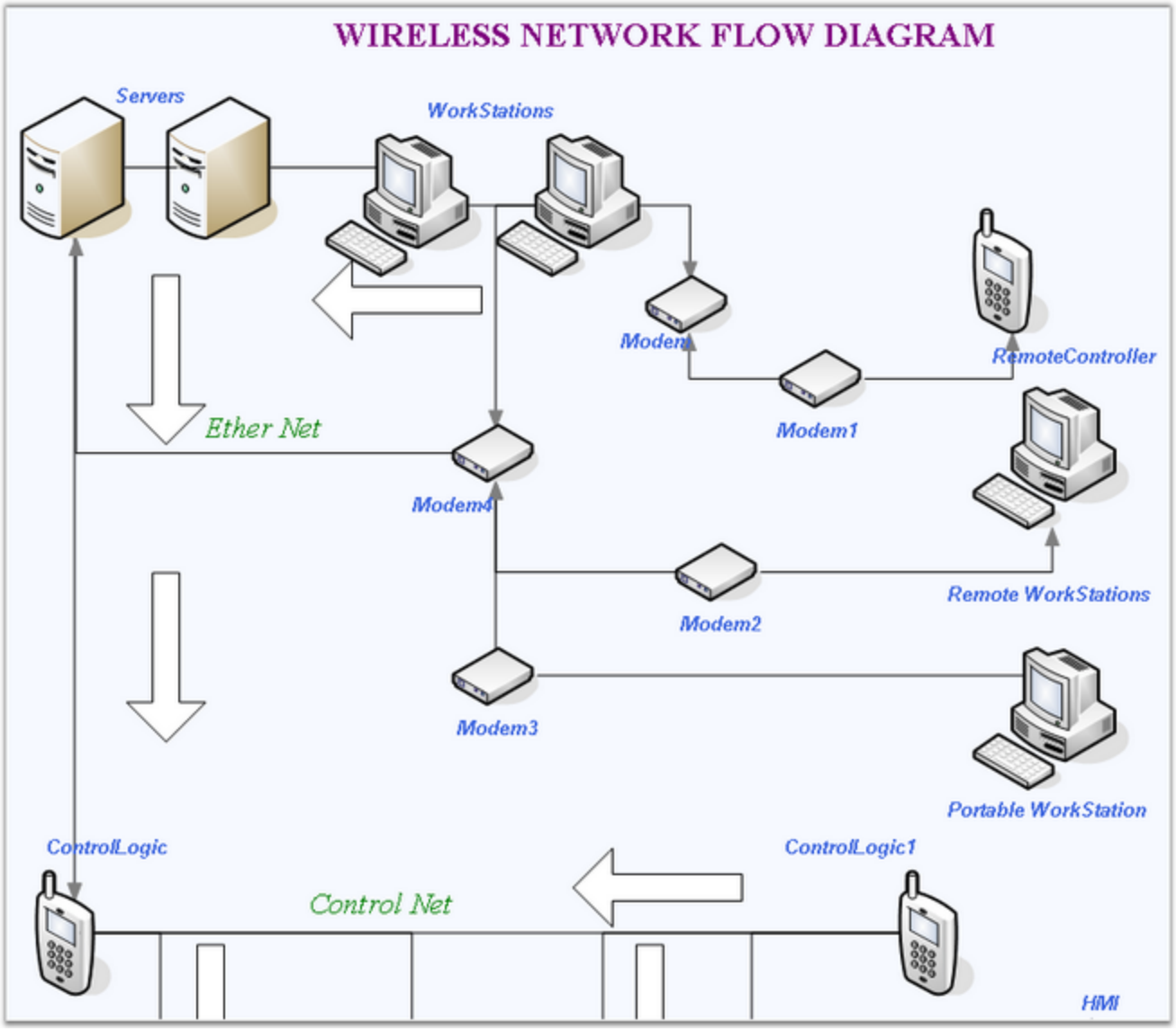

Figure 1: Wireless Network flow

Key Features

Some of the key features of Essential Diagram are listed below.

- Essential Diagram supports Matrix Transformations like Translate (Move), Rotate and Scale.
- Shape nodes are the graphical objects that can be drawn on the diagram area by activating one of several drawing tools such as the RectangleTool, RoundRectTool, EllipseTool, LineTool, PolylineTool, OrthogonallineTool, BezierTool, CurveTool, ArcTool and PolygonTool.
- DecoratorShape can be added at the head and tail of Connectors. The shapes include arrows, circles, diamonds, crosses, squares and custom decorators.

- Essential Diagram supports rendering Textnodes and RichTextnodes, and offers full text formatting through sufficient properties. Text editing and text rotation are also supported.
- Zooming, scrolling, and panning are supported and can be achieved using sufficient interactive diagram tools.
- Automatic Line Routing and Line Bridging: while a link is drawn between two nodes and if any other node is found in between them, the line will be automatically re-routed around those nodes.
- The PrintDialog class enables the user to set the printer to be used, and allows to define the pages and the number of copies to be printed.
- The PrintPreviewDialog class provides an overview of how the document will appear when printed, which is invoked using the ShowDialog method.
- PaletteGroupBar control is a WinForm control that can be added to the Visual Studio .NET toolbox. It has the following capabilities:

    - It displays list of symbols in a symbol palette as icons
    - It allows user to drag symbols onto diagrams
    - It supports multiple symbol palettes at a time
    - It has a user interface similar to Microsoft Outlook bar
    - It is implemented based on the Syncfusion GroupBar and GroupView controls

User Guide Organization

The product comes with numerous samples as well as an extensive documentation to guide you. This User Guide provides detailed information on the features and functionalities of the Tools controls. It is organized into the following sections:

- **Overview**-This section gives a brief introduction to our product and its key features.
- **Installation and Deployment**-This section elaborates on the install location of the samples, license etc.
- **What's New**-This section lists the new features implemented for every release.
- **Getting Started**-This section guides you on getting started with Windows application, controls etc.
- **Concepts and Features**-The features of Essential Diagram are illustrated with use case scenarios, code examples and screen shots under this section.

Document Conventions

The conventions listed below will help you to quickly identify the important sections of information, while using the content:

Convention	Icon	Description
Note	***Note:***	Represents important information
Example	**Example**	Represents an example

Tip		Represents useful hints that will help you in using the controls/features
Additional Information		Represents additional information on the topic

1.2 Prerequisites and Compatibility

This section covers the requirements mandatory for using Essential Diagram control. It also lists operating systems and browsers compatible with the product.

Prerequisites

The prerequisites details are listed below:

Development Environments	• Visual Studio 2013 • Visual Studio 2012 • Visual Studio 2010 (Ultimate and Express) • Visual Studio 2008 (Team, Professional, Standard and Express) • Visual Studio 2005 (Team, Professional, Standard and Express) • Borland Delphi for .NET • SharpCode
.NET Framework versions	• .NET 4.5 • .NET 4.0 • .NET 3.5 • .NET 2.0

Compatibility

The compatibility details are listed below:

Operating Systems	• Windows 8.1 • Windows Server 2008 (32 bit and 64 bit) • Windows 7 (32 bit and 64 bit) • Windows Vista (32 bit and 64 bit) • Windows XP • Windows 2003

1.3 Documentation

Syncfusion provides the following documentation segments to provide all necessary information for using Essential Diagram control in Windows application in an efficient manner.

Type of documentation	Location
Readme	[drive:]\Program Files\Syncfusion\Essential Studio\x.x.x.x\Infrastructure\Data\Release Notes\readme.htm
Release Notes	[drive:]\Program Files\Syncfusion\Essential Studio\x.x.x.x\Infrastructure\Data\Release Notes\Release Notes.htm
User Guide (this document)	**Online** http://help.syncfusion.com/resources(Navigate to the Diagram for Windows Forms User Guide.)  ***Note: Click Download as PDF to access a PDF version.*** **Installed Documentation** Dashboard -> Documentation -> Installed Documentation.
Class Reference	**Online** http://help.syncfusion.com/resources(Navigate to the Windows Forms User Guide. Select *Diagram* in the second text box, and then click the Class Reference link found in the upper right section of the page.) **Installed Documentation** Dashboard -> Documentation -> Installed Documentation.

2 Installation And Deployment

This section covers information on the install location, samples, licensing, patches update and updation of the recent version of Essential Studio. It comprises the following sub-sections:

2.1 Installation

For step-by-step installation procedure for the installation of Essential Studio, refer to the **Installation** topic under **Installation and Deployment** in the **Common UG**.

See Also

For licensing, patches and information on adding or removing selective components, refer the following topics in Common UG under **Installation and Deployment**.

- Licensing
- Patches
- Add / Remove Components

2.2 Sample And Location

This section covers the location of the installed samples and describes the procedure to run the samples through the sample browser. It also lists the location of source code.

Sample Installation Location

The Essential Diagram Windows Forms samples are installed in the following location.

...\My Documents\Syncfusion\EssentialStudio\Version Number\Windows\Diagram.Windows\Samples\2.0

Viewing Samples

To view the samples, follow the steps below:

1. Click Start-->All Programs-->Syncfusion-->Essential Studio <version number> -->Dashboard.

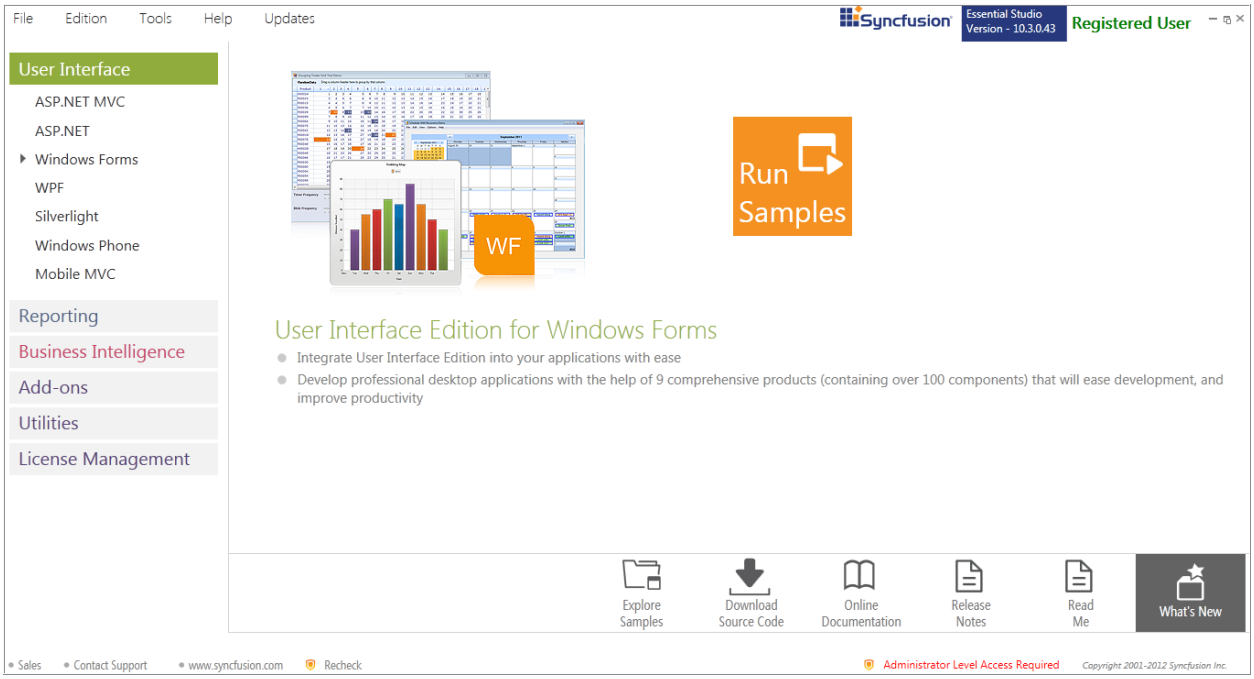

Figure 2: Essential Studio Dashboard

2. In the Dashboard window, click **Run Samples** for Windows Forms under UI Edition. The UI Windows Form Sample Browser window is displayed.

Note: You can view the samples in any of the following three ways:

- **Run Samples**-Click to view the locally installed samples
- **Online Samples**-Click to view online samples
- **Explore Samples**-Explore BI Web samples on disk

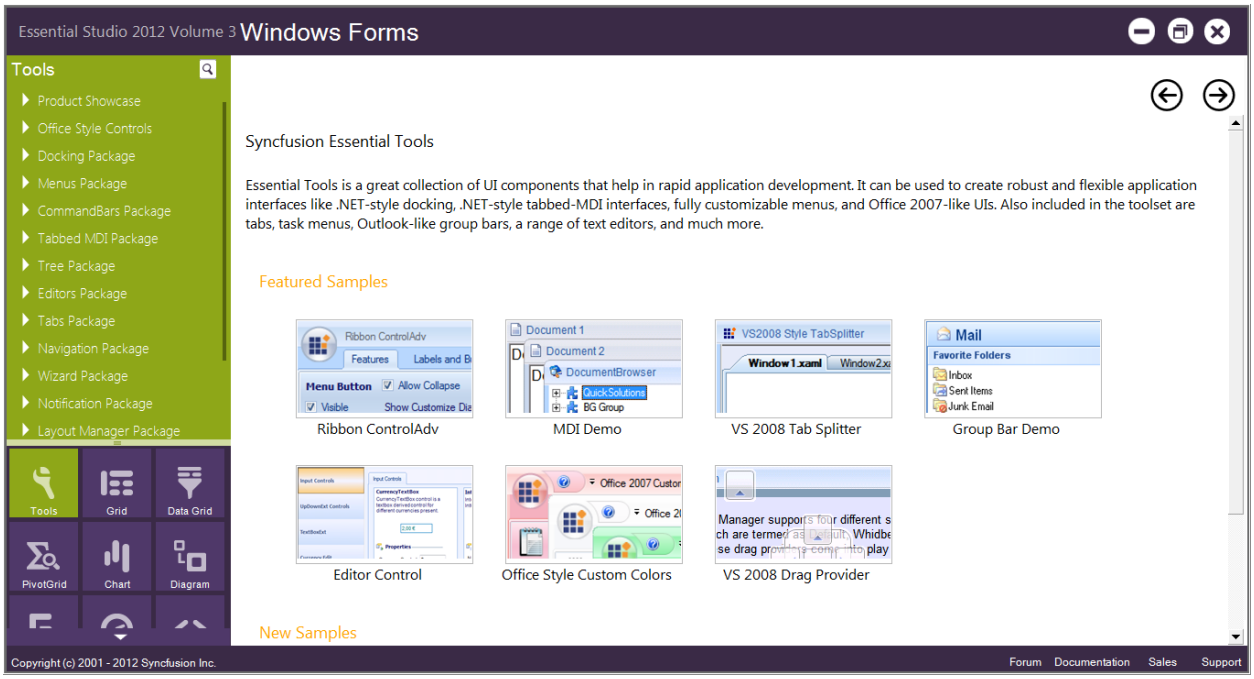

Figure 3: User Interface Edition Windows Forms Sample Browser

3. To view the samples of Diagram control, click **Diagram** from the bottom-left pane.

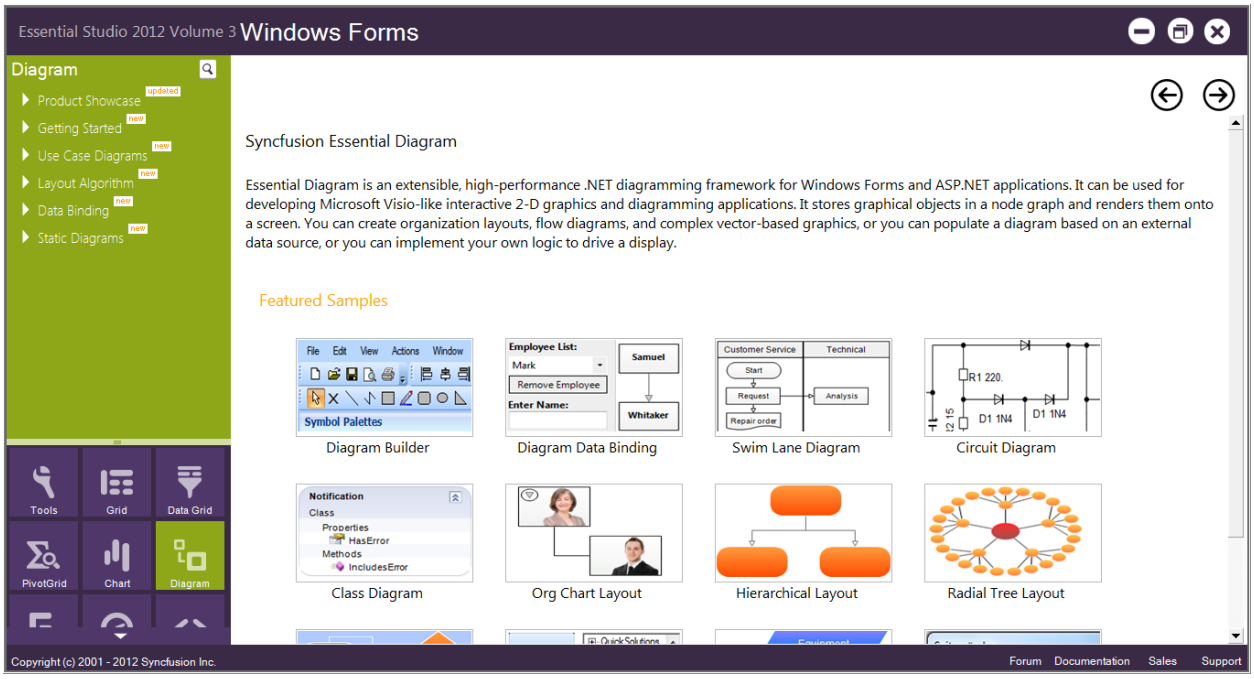

Figure 4: Diagram Samples for Windows

4. Select any sample and browse through the features.

Source Code Location

The source code for Essential Diagram Windows is available at the following default location:

[System Drive]:\Program Files\Syncfusion\Essential Studio\[Version Number]\Windows\Diagram.Windows\Src

2.3 Deployment Requirements

This section provides deployment requirements for using Essential Diagram under the following topics:

2.3.1 Toolbox Entries

Essential Diagram places the following controls into your Visual Studio .NET toolbox from where you can drag each control onto a form and start working with it.

- Diagram

2.3.2 DLLs

The following assemblies need to be referenced in your application for using Diagram control:

Windows Forms – Diagram(Basic)

- Syncfusion.Core.dll
- Syncfusion.Shared.Base.dll
- Syncfusion.Diagram.Base.dll
- Syncfusion.Diagram.Windows.dll

Windows Forms – Diagram(On inclusion of Scripting)

- Syncfusion.Core.dll
- Syncfusion.Shared.Base.dll
- Syncfusion.Diagram.Base.dll
- Syncfusion.Diagram.Windows.dll
- Syncfusion.Scripting.Base
- Syncfusion.Scripting.Windows

3 Getting Started

This section helps you to understand and quickly get started using Essential Diagram in your Windows Forms applications. Control appearance and structure are defined, and the relevant classes are depicted.

3.1 Structure of Essential Diagram

The Essential Diagram package is comprised of the following controls:

- Diagram
- Overview
- PaletteGroupBar
- PaletteGroupView
- PropertyEditor
- DocumentExplorer

Diagram

The Diagram control is an interactive two-dimensional graphics control for diagramming, technical drawing, visualization, and simulation applications. It provides a surface for rendering and manipulating 2-D shapes, text, images, and Windows Form controls. The UI supports drag-and-drop, scaling, zooming, rotation, grouping, ungrouping, connection points, layouts, and many other features.

The Diagram control's architecture is composed of three objects, namely **Model**, **View**, and **Controller** and it provides a clear separation between data, visualization, and user interface. One advantage of model-view-controller architecture is that the parts are interchangeable. The model, view, and controller can be swapped in and out independently.

The model contains the data (node/connector) portion of a diagram. The view is responsible for rendering the diagram, and the controller handles user interaction. The Diagram control supports both horizontal and vertical rulers.

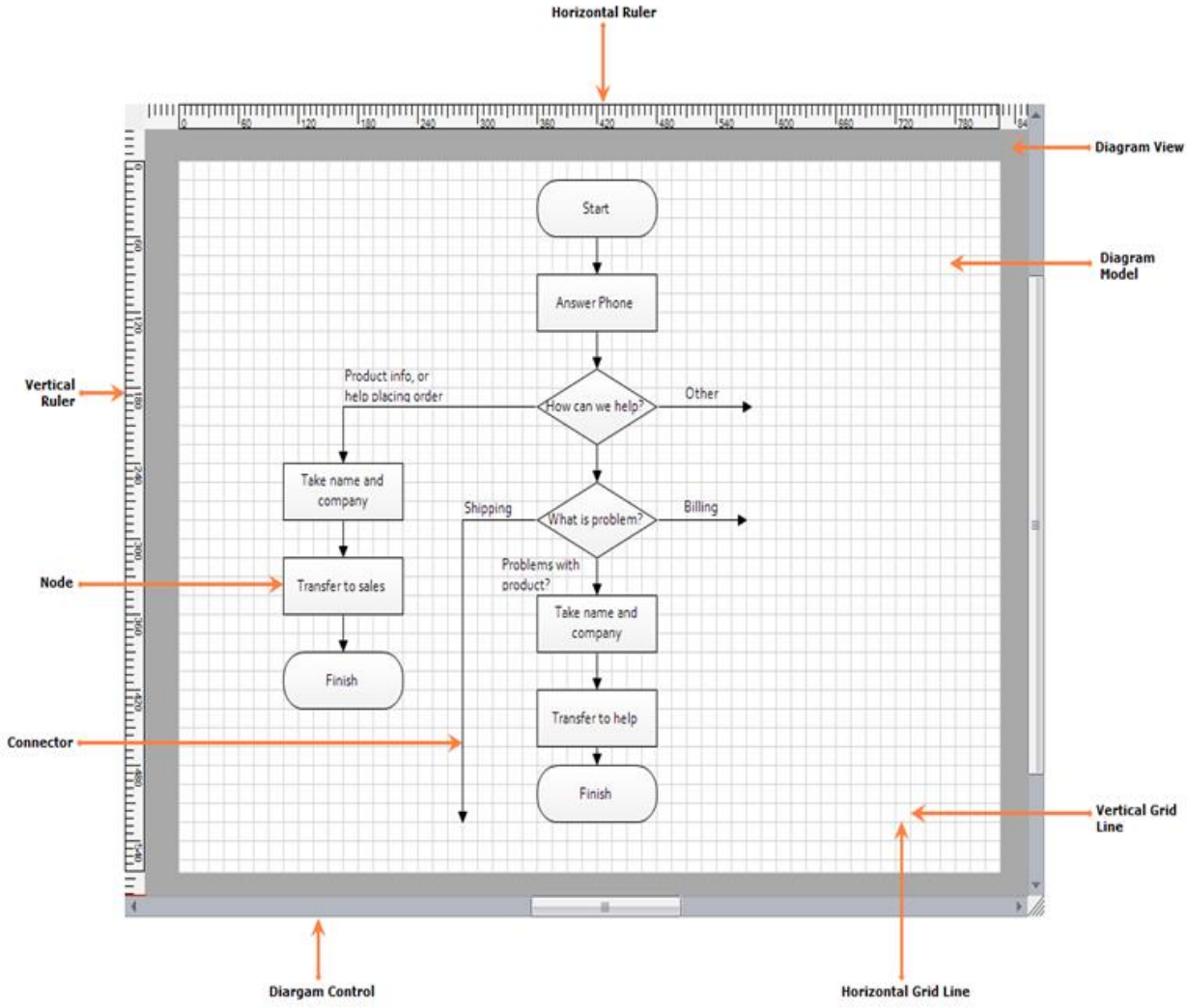

Figure 5: Diagram Control

Overview

The Overview control provides a perspective view of a diagram model and allows users to dynamically pan/zoom the diagram. It features a viewport window that can be moved and/or resized using the mouse to modify the diagram's origin and magnification properties at run time.

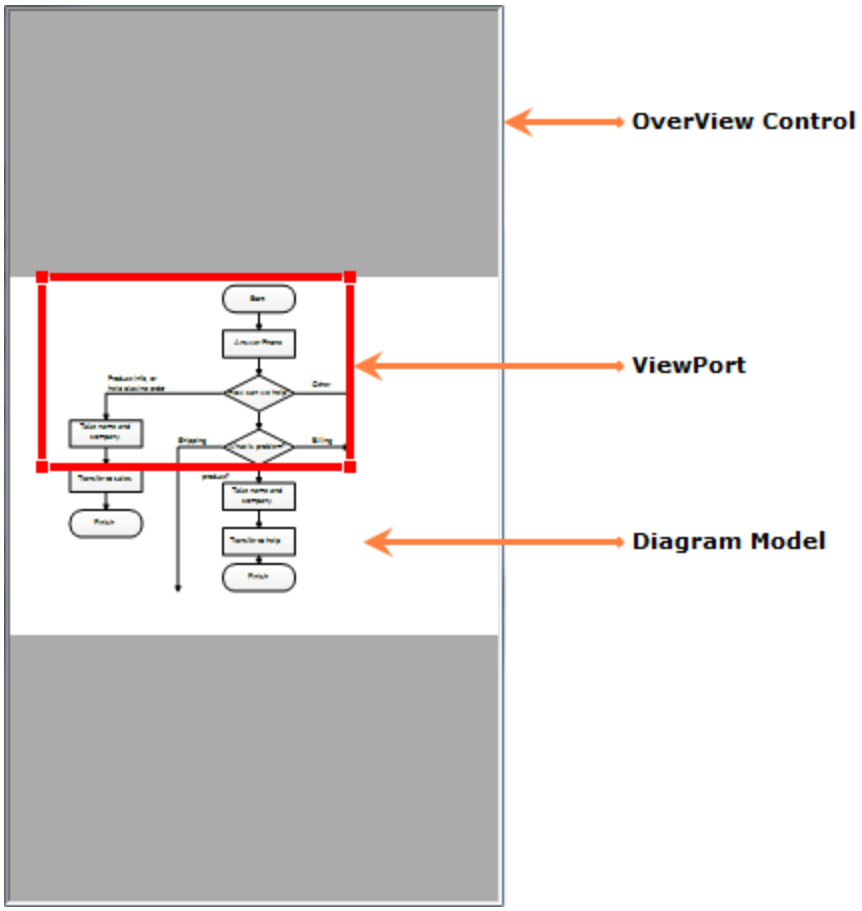

Figure 6: Overview Control

PaletteGroupBar

The PaletteGroupBar control provides a way for users to drag symbols onto a diagram. It is based on the GroupBar control of Essential Tools. Each symbol palette loaded in the PaletteGroupBar control occupies a panel that can be selected by a bar button. The bar button is labeled with the name of the symbol palette. Each symbol palette is a list of symbols that have an icon and a label. The symbols in the palette are shown as icons that can be dragged onto the diagram. This control allows users to add symbols to a palette, and save or load the palette whenever necessary. It provides a way to classify and maintain symbols. It also provides preview of symbol during drag-and-drop operation.

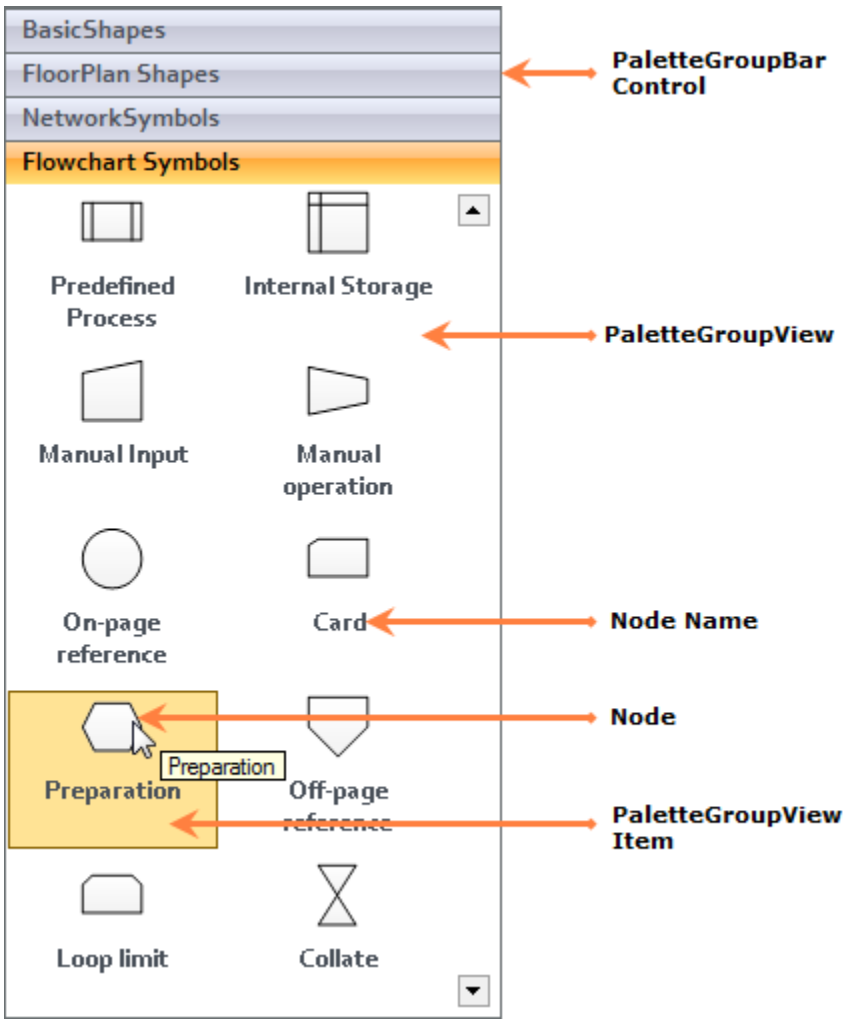

Figure 7: PaletteGroupBar Control

PaletteGroupView

The PaletteGroupView control provides an easy way to serialize a symbol palette to and from a resource file of a form. At design time, users can attach a symbol palette to the PaletteGroupView control in the form. Selecting **PaletteGroupView** and clicking the **Palette** property in the Visual Studio .NET Properties window will open a standard **Open File** dialog, which allows the user to select a symbol palette file that has been created with the Symbol Designer.

It displays the symbol models belonging to a symbol palette in the GroupView control of Essential Tools for Windows Froms. It contains a list of icons and labels that can be hosted in the PaletteGroupBar control. The symbol models can be dragged from this control and dropped onto the diagram. It also provides preview of symbols during drag-and-drop operation.

PropertyEditor

The PropertyEditor control displays and edits the properties of diagram models, and nodes on the diagram. It contains an embedded PropertyGrid control which is used to edit the properties of selected objects on the diagram. A combo box is added to this control for listing out the names of objects and selecting the objects on the diagram.

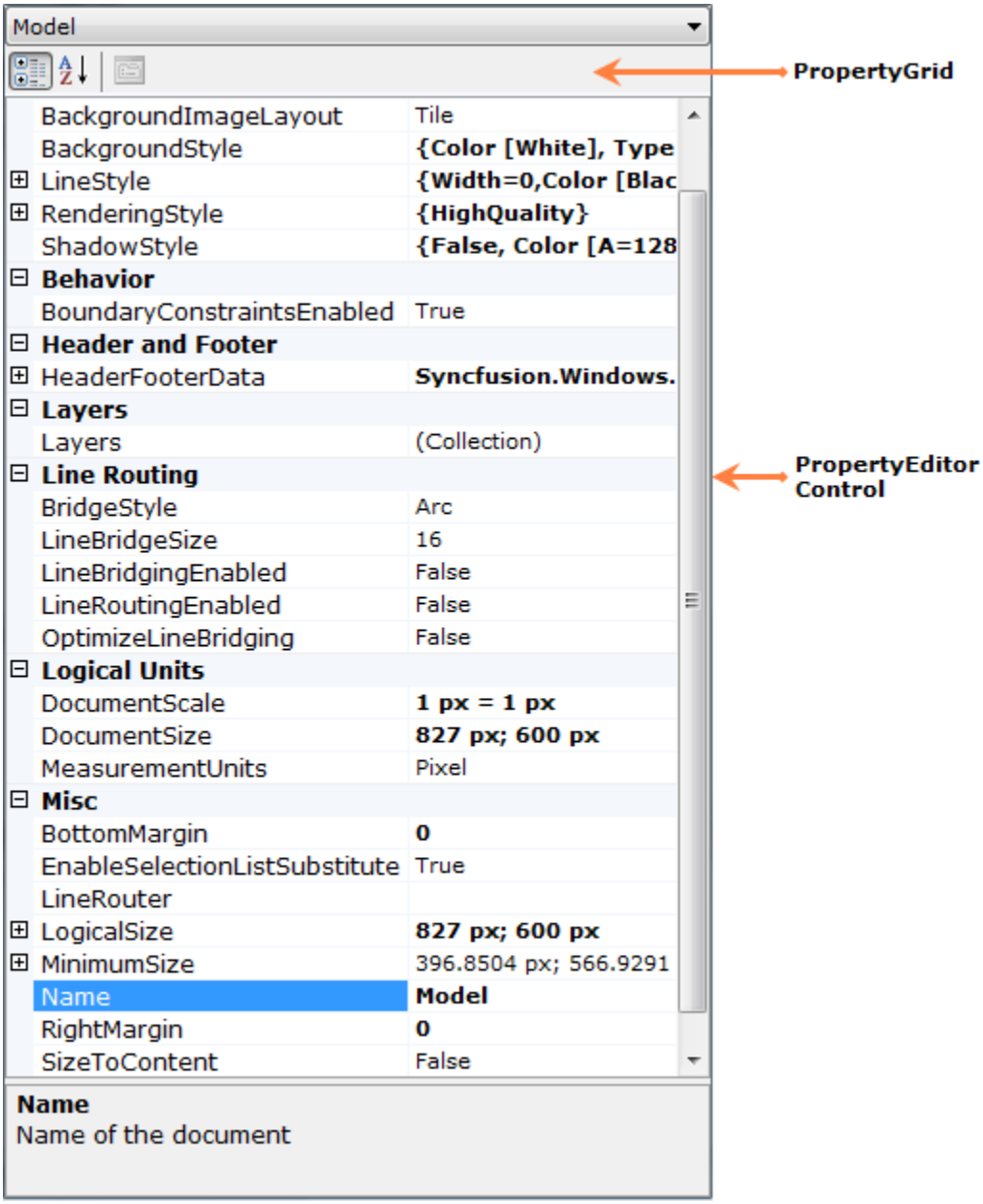

Figure 8: PropertyEditor Control

DocumentExplorer

The DocumentExplorer control allows you to visualize the details of the various objects that are added onto the diagram model. The layers will be listed under the **Layers** node, and other objects like shapes, links, lines, and text editors will be listed under the **Nodes** node. The nodes and layers can be renamed, deleted, and hided, and a new layer can be added through this control.

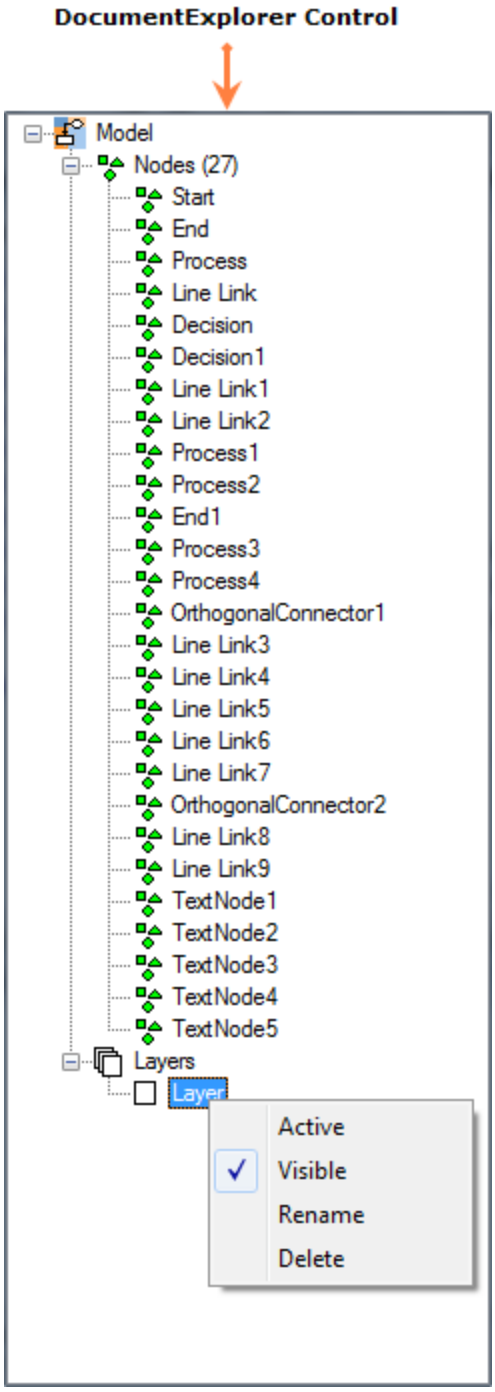

Figure 9: DocumentExplorer Control

3.2 Essential Diagram in Windows Forms Application

This section helps you to create the Diagram, PaletteGroupBar, PaletteGroupView, Overview, PropertyEditor, and DocumenExplorer controls through the designer and code in a Windows Forms application.

3.2.1 Diagram

3.2.1.1 Creating a Diagram Control through Designer

This section depicts the step-by-step procedure to create a Diagram control through the Visual Studio designer in a .NET Windows Forms application.

To create a Diagram control using the designer:

1. Create a new Windows Forms application.
2. Open the **Designer Form** window.
3. Drag **Diagram** from the **Toolbox** window and drop it to the **Designer Form** window.

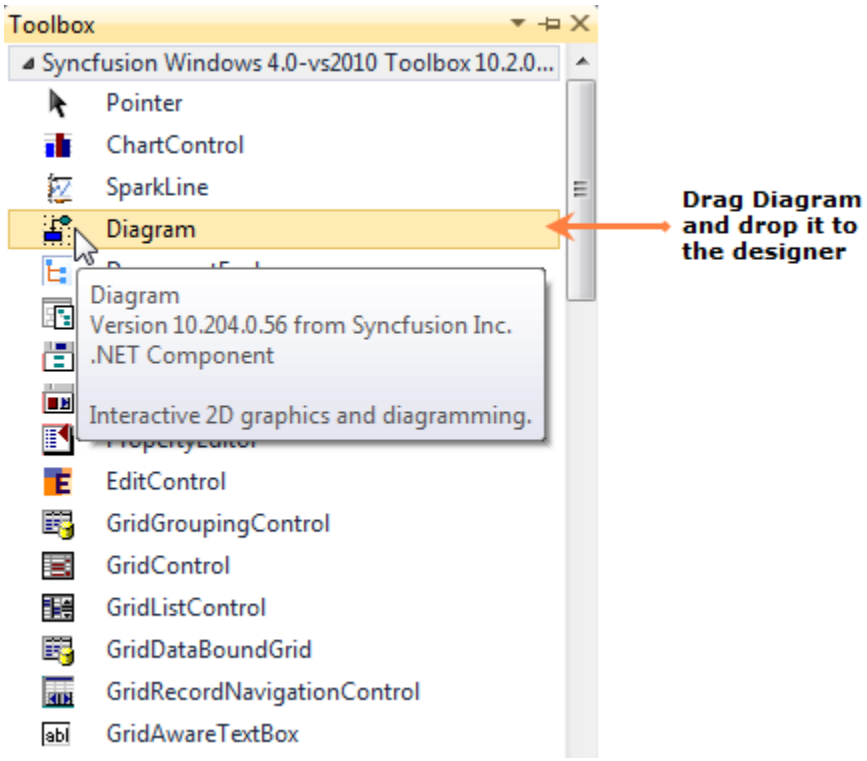

Figure 10: Dragging Diagram to the Designer

The Diagram control will be added to the designer and its dependent assemblies will be added to the project once you dropped it to the Designer Form window.

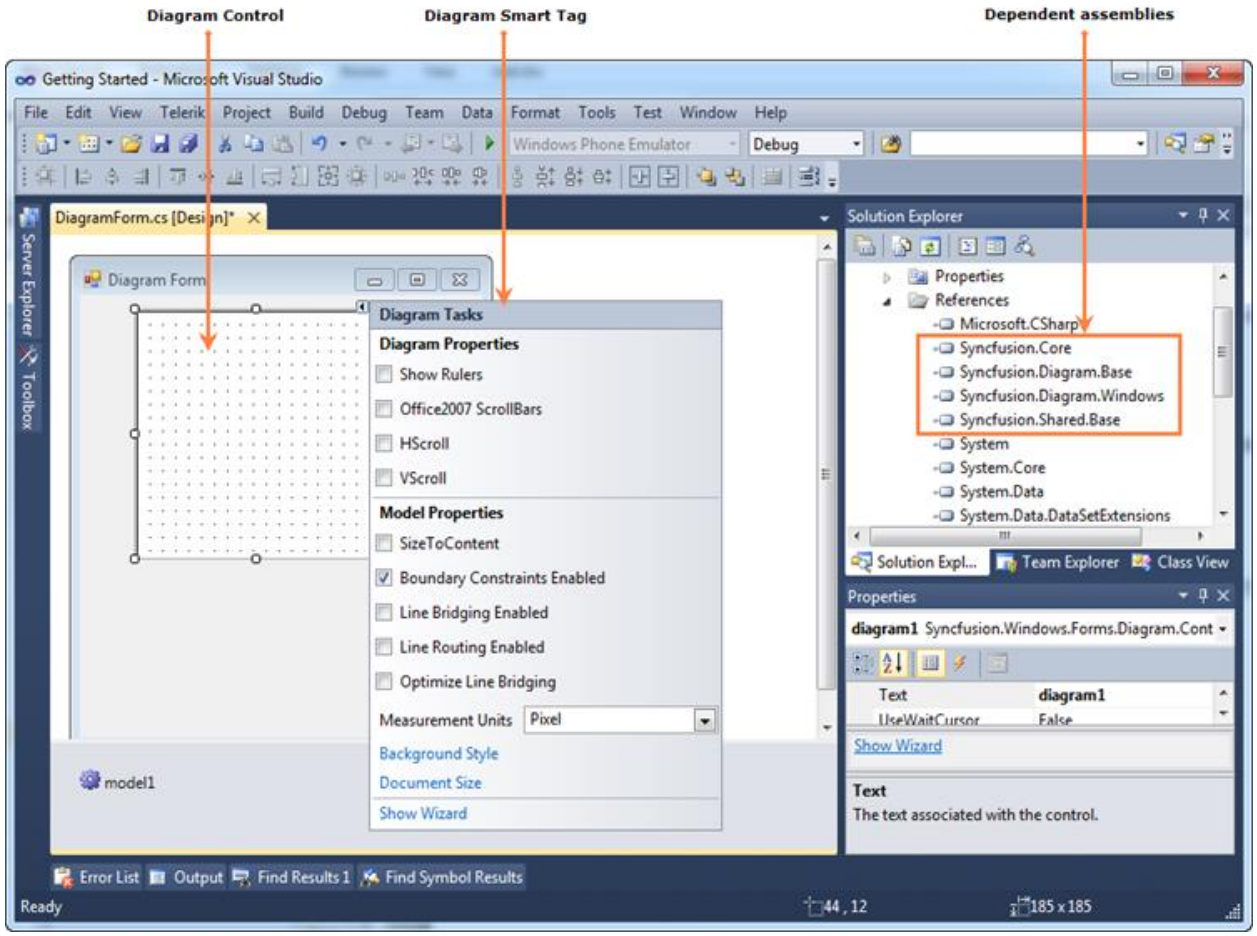

Figure 11: Designer Form Window Added with Diagram Control

3.2.1.2 Creating a Diagram Control through Code

This section shows the step-by-step procedure to create a Diagram control programmatically in a .NET Windows Forms application.

To create a Diagram control using code:

1. Create a new Windows Forms application.
2. Add the following basic dependent Syncfusion assemblies to the project:
   - Syncfusion.Core.dll
   - Syncfusion.Diagram.Base.dll
   - Syncfusion.Diagram.Windows.dll
   - Syncfusion.Shared.Base.dll

3. Create a Diagram control using the following code.

[C#]

```
//Imports the Diagram control's namespaces
using Syncfusion.Windows.Forms.Diagram.Controls;
using Syncfusion.Windows.Forms.Diagram;

//Create an instance
Diagram diagram = new Diagram();

//Enable scroll bars
diagram.HScroll = true;
diagram.VScroll = true;

//Sizing the diagram
diagram.Size = new Size(400, 400);
//Positioning the diagram
diagram.Location = new Point(20, 5);
```

[VB]

```
'Imports the Diagram control's namespaces
Imports Syncfusion.Windows.Forms.Diagram
Imports Syncfusion.Windows.Forms.Diagram.Controls

'Create an instance
Dim diagram As New Diagram()

'Enable Scrollbars
diagram.HScroll = True
diagram.VScroll = True

'Sizing the diagram
diagram.Size = New Size(400, 400)
'Positioning the diagram
```

```
            diagram.Location = New Point(20, 5)
```

4. Add a model to the Diagram control.

```
[C#]
            //Create a model
            Model model = new Model();

            //Add the model to the Diagram control
            diagram.Model = model;
```

```
[VB]
            'Create a model
            Dim model As New Model()

            'Add the model to the Diagram control
            diagram.Model = model
```

5. Add the Diagram control to the **Diagram Form** window.

```
[C#]
            //Add the Diagram control to Diagram Form
            this.Controls.Add(diagram);
```

```
[VB]
            'Add the Diagram control to the Diagram Form
            Me.Controls.Add(diagram)
```


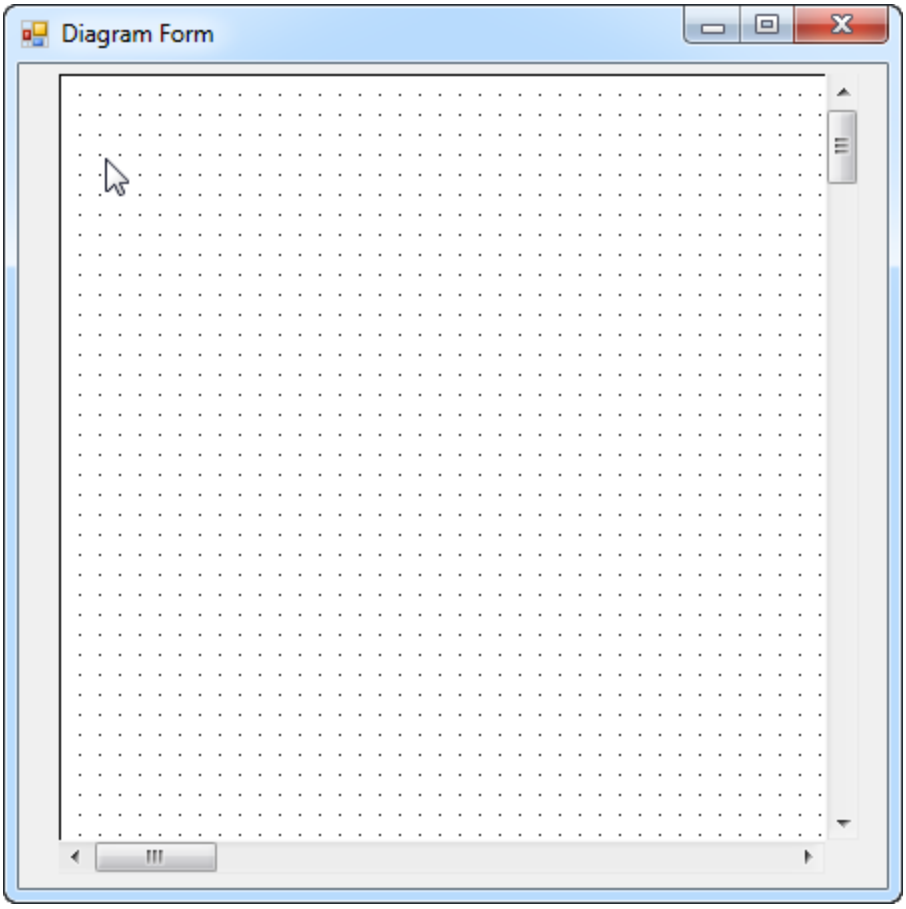

Figure 12: Diagram Control Created using Code

3.2.1.3 Adding Nodes to the Model

The Diagram control has a list of predefined basic shapes (nodes) which help you to draw diagrams according to your requirement. You can create your own shapes by inheriting the existing shape's class and the Symbol Designer utility tool which is shipped with the Essential Diagram package.

The following code creates a rectangular node and adds it to the model.

```
[C#]
            //Enable diagram rulers
            diagram.ShowRulers = true;


            //Create a rectangular node
```



```
            Syncfusion.Windows.Forms.Diagram.Rectangle rectangle
                = new Syncfusion.Windows.Forms.Diagram.Rectangle(120,
120, 100, 70);

            //Style the rectangular node
            rectangle.FillStyle.Type = FillStyleType.LinearGradient;
            rectangle.FillStyle.Color = Color.FromArgb(128, 0, 0);
            rectangle.FillStyle.ForeColor = Color.FromArgb(225, 0, 0);
            rectangle.ShadowStyle.Visible = true;

            //Border style
            rectangle.LineStyle.LineColor = Color.RosyBrown;
            rectangle.LineStyle.LineWidth = 2.0f;
            rectangle.LineStyle.LineJoin = LineJoin.Miter;

            //Add a label to the rectangular node
             Syncfusion.Windows.Forms.Diagram.Label label
                = new Syncfusion.Windows.Forms.Diagram.Label();
            label.Text = "Hello!";
            label.FontStyle.Family = "Arial";
            label.FontColorStyle.Color = Color.White;
            rectangle.Labels.Add(label);

            //Add the rectangular node to the model
            diagram.Model.AppendChild(rectangle);
```

[VB]

```
            'Enable diagram rulers
            diagram.ShowRulers = True

            'Create a rectangular node
            Dim rectangle As New Rectangle(120, 120, 100, 70)
```



```
'Style the rectangular node
rectangle.FillStyle.Type = FillStyleType.LinearGradient
rectangle.FillStyle.Color = Color.FromArgb(128, 0, 0)
rectangle.FillStyle.ForeColor = Color.FromArgb(225, 0, 0)
rectangle.ShadowStyle.Visible = True

'Border style
rectangle.LineStyle.LineColor = Color.RosyBrown
rectangle.LineStyle.LineWidth = 2.0F
rectangle.LineStyle.LineJoin = Drawing2D.LineJoin.Miter

'Add a label to the rectangular node
Dim label As New Syncfusion.Windows.Forms.Diagram.Label()
label.Text = "Hello!"
label.FontStyle.Family = "Arial"
label.FontColorStyle.Color = Color.White
rectangle.Labels.Add(label)

'Add the rectangular node to the model
diagram.Model.AppendChild(rectangle)
```

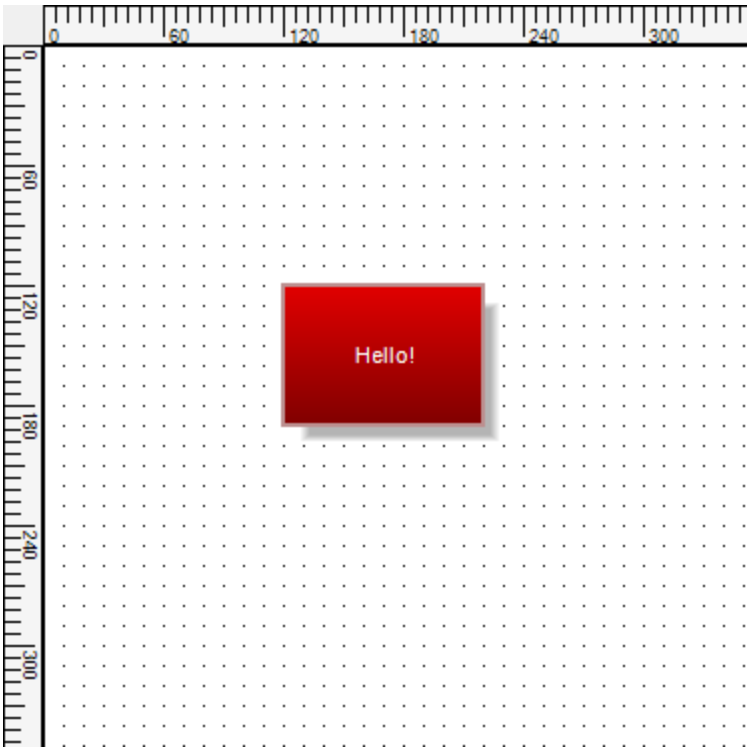

Figure 13: Rectangular Node

3.2.1.4 Connecting Nodes

The Diagram control has a set of predefined links (connectors) which help you to connect the nodes with each other. You can create your own connectors by inheriting the **ConnectorBase** class.

The following code illustrates how to connect a process node to a decision node by **OrthogonalConnector**.

```
[C#]
             //Create a process node
            Syncfusion.Windows.Forms.Diagram.Rectangle process
                = new Syncfusion.Windows.Forms.Diagram.Rectangle(50,
50, 100, 70);


            //Style the process node
            process.FillStyle.Type = FillStyleType.LinearGradient;
            process.FillStyle.Color = Color.FromArgb(128, 0, 0);
```



```
            process.FillStyle.ForeColor = Color.FromArgb(225, 0, 0);

            //Border style
            process.LineStyle.LineColor = Color.RosyBrown;
            process.LineStyle.LineWidth = 2.0f;
            process.LineStyle.LineJoin = LineJoin.Miter;

            //Add a label to the process node
            Syncfusion.Windows.Forms.Diagram.Label label
                = new Syncfusion.Windows.Forms.Diagram.Label();
            label.Text = "Process";
            label.FontStyle.Family = "Arial";
            label.FontColorStyle.Color = Color.White;
            process.Labels.Add(label);

            //Add the process node to the model
            diagram.Model.AppendChild(process);

            //Create a decision node
            Polygon decision = new Polygon(new PointF[] {
                            new PointF(0,50), new PointF(50,0),
                            new PointF(100,50), new PointF(50,100),
                            new PointF(0,50)});

            //Style the decision node
            decision.FillStyle.Type = FillStyleType.LinearGradient;
            decision.FillStyle.Color = Color.FromArgb(128, 0, 0);
            decision.FillStyle.ForeColor = Color.FromArgb(225, 0, 0);

            //Border style
            decision.LineStyle.LineColor = Color.RosyBrown;
            decision.LineStyle.LineWidth = 2.0f;
            decision.LineStyle.LineJoin = LineJoin.Miter;
```

```
            //Add a label to the decision node
            label = new Syncfusion.Windows.Forms.Diagram.Label();
            label.Text = "Decision";
            label.FontStyle.Family = "Arial";
            label.FontColorStyle.Color = Color.White;
            decision.Labels.Add(label);

            //Position the decision node
            decision.PinPoint = new PointF(250, 250);
            //Add decision node to the Model
            diagram.Model.AppendChild(decision);

            //Create an orthogonal connector
            OrthogonalConnector link =
                new OrthogonalConnector(process.PinPoint,
decision.PinPoint);

            //Style the link
            link.LineStyle.LineColor = Color.RosyBrown;
            link.LineStyle.LineWidth = 2f;

            //Head decorator style
            link.HeadDecorator.DecoratorShape =
DecoratorShape.Filled45Arrow;
            link.HeadDecorator.Size = new SizeF(8, 8);
            link.HeadDecorator.FillStyle.Color = Color.RosyBrown;
            link.HeadDecorator.LineStyle.LineColor = Color.RosyBrown;

            //Connect a tail node to a head node
            process.CentralPort.TryConnect(link.TailEndPoint);
//process is tail node
            decision.CentralPort.TryConnect(link.HeadEndPoint);
//decision is head node

            //Add a link to the model
```



```
            diagram.Model.AppendChild(link);
```

[VB]

```
            'Create a process node
            Dim process As New Rectangle(50, 50, 100, 70)

            'Style the process node
            process.FillStyle.Type = FillStyleType.LinearGradient
            process.FillStyle.Color = Color.FromArgb(128, 0, 0)
            process.FillStyle.ForeColor = Color.FromArgb(225, 0, 0)

            'Border style
            process.LineStyle.LineColor = Color.RosyBrown
            process.LineStyle.LineWidth = 2.0F
            process.LineStyle.LineJoin = Drawing2D.LineJoin.Miter

            'Add a label to the process node
            Dim label As New Syncfusion.Windows.Forms.Diagram.Label()
            label.Text = "Process"
            label.FontStyle.Family = "Arial"
            label.FontColorStyle.Color = Color.White
            process.Labels.Add(label)

            'Add process node to the model
            diagram.Model.AppendChild(process)

            'Create a decision node
            Dim decision As New Polygon(
                New PointF() {
                New PointF(0, 50),
                New PointF(50, 0),
                New PointF(100, 50),
```



```
                New PointF(50, 100),
                New PointF(0, 50)
               })

            'Style the decision node
            decision.FillStyle.Type = FillStyleType.LinearGradient
            decision.FillStyle.Color = Color.FromArgb(128, 0, 0)
            decision.FillStyle.ForeColor = Color.FromArgb(225, 0, 0)

            'Border style
            decision.LineStyle.LineColor = Color.RosyBrown
            decision.LineStyle.LineWidth = 2.0F
            decision.LineStyle.LineJoin = Drawing2D.LineJoin.Miter

            'Add a label to the decision node
            label = New Syncfusion.Windows.Forms.Diagram.Label()
            label.Text = "Decision"
            label.FontStyle.Family = "Arial"
            label.FontColorStyle.Color = Color.White
            decision.Labels.Add(label)

            'Position the decision node
            decision.PinPoint = New PointF(250, 250)
            'Add decision node to the model
            diagram.Model.AppendChild(decision)

            'Create an orthogonal connector
            Dim link As New OrthogonalConnector(process.PinPoint,
decision.PinPoint)

            'Style the link
            link.LineStyle.LineColor = Color.RosyBrown
            link.LineStyle.LineWidth = 2.0F
```

```
            'Head decorator style
            link.HeadDecorator.DecoratorShape =
DecoratorShape.Filled45Arrow
            link.HeadDecorator.Size = New SizeF(8, 8)
            link.HeadDecorator.FillStyle.Color = Color.RosyBrown
            link.HeadDecorator.LineStyle.LineColor = Color.RosyBrown

            'Connect a tail node to a head node
            process.CentralPort.TryConnect(link.TailEndPoint) 'process
is tail node
            decision.CentralPort.TryConnect(link.HeadEndPoint)
'decision is head node

            'Add a link to the model
            diagram.Model.AppendChild(link)
```

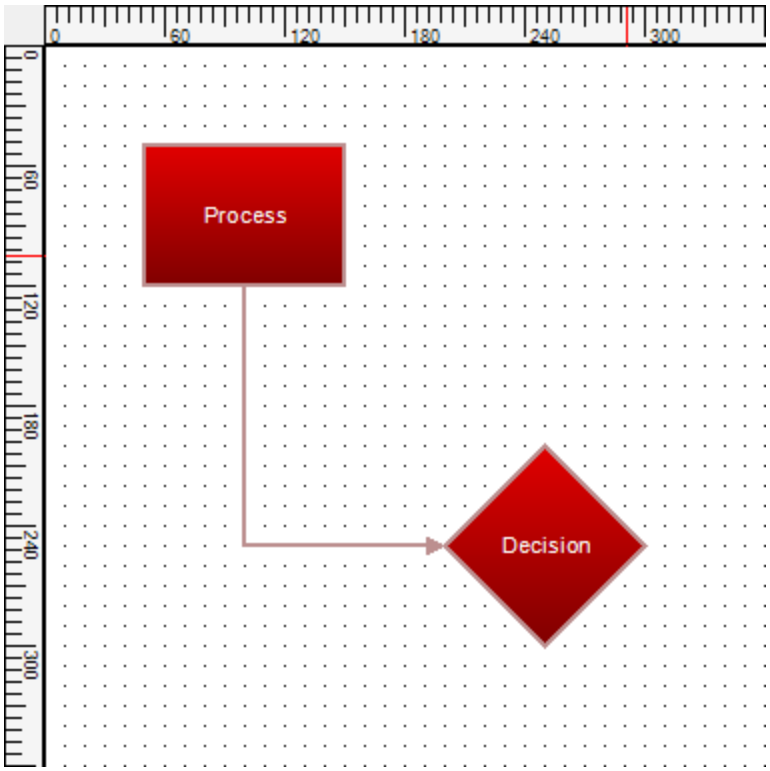

Figure 14: Connecting Nodes

3.2.2 PaletteGroupBar

3.2.2.1 Creating a PaletteGroupBar Control through Designer

This section depicts the step-by-step procedure to create a PaletteGroupBar control through the Visual Studio designer in a .NET Windows Forms application.

To create a PaletteGroupBar control through the designer:

1. Create a new Windows Forms application.
2. Open the **Designer Form** window.
3. Drag **PaletteGroupBar** from **Toolbox** and drop it to the **Designer Form** window.

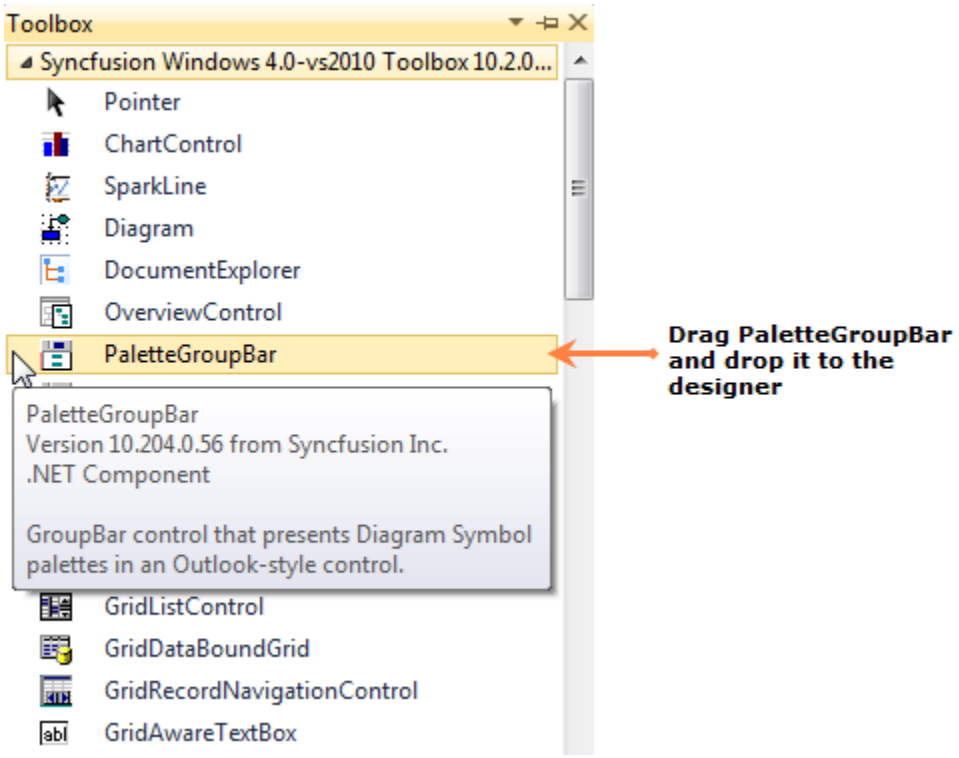

Figure 15: Dragging PaletteGroupBar to the Designer

The PaletteGroupBar control will be added to the designer and its dependent assemblies will be added to the project once you dropped it to the Designer Form window.

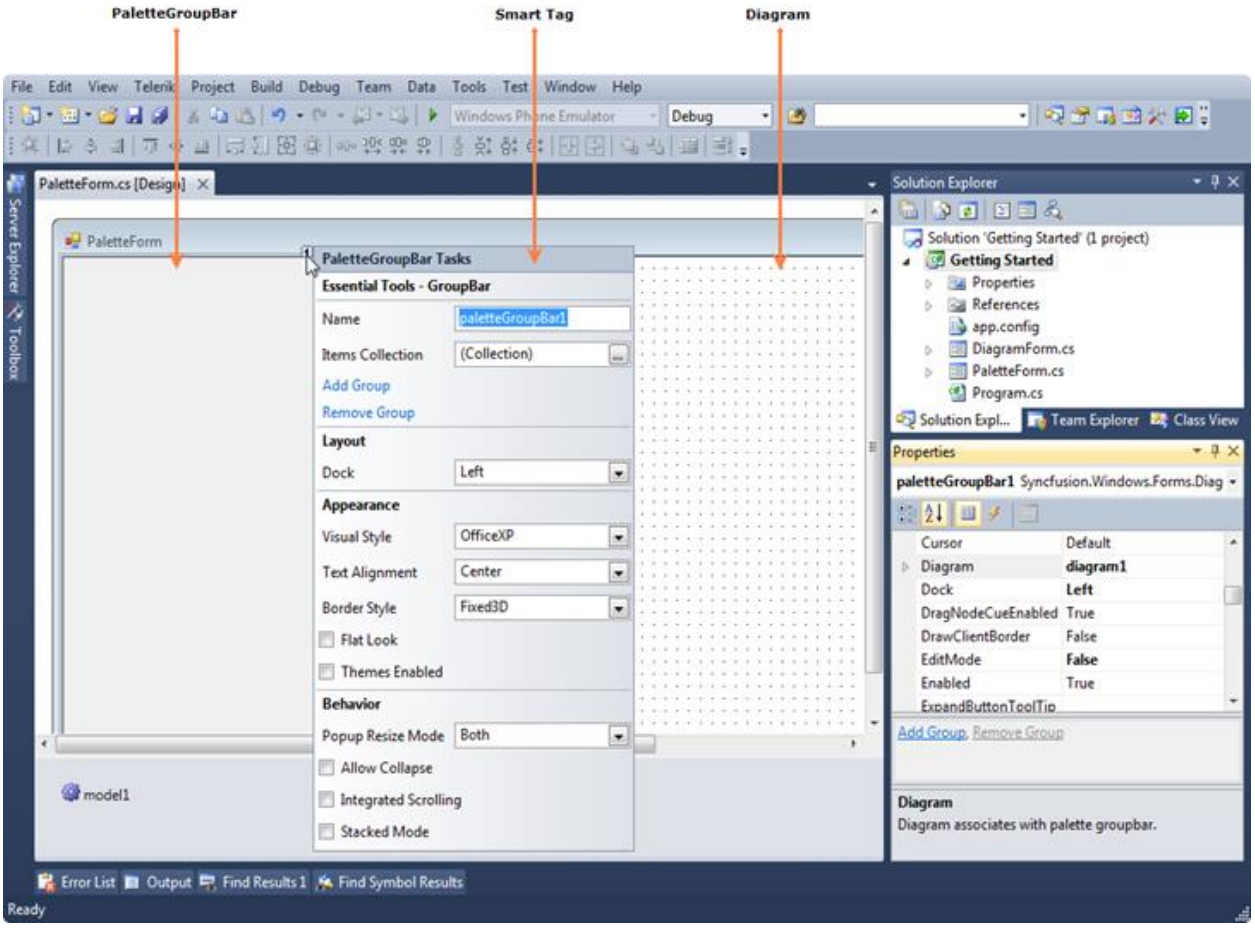

Figure 16: Designer Form Window Added with PaletteGroupBar Control

3.2.2.2 Creating a PaletteGroupBar Control through code

This section shows the step-by-step procedure to create a PaletteGroupBar control programmatically in a .NET Windows Forms application.

To create a PaletteGroupBar control using code:

1. Create a new Windows Forms application.
2. Add the following basic dependent Syncfusion assemblies to the project:
   - Syncfusion.Core.dll
   - Syncfusion.Diagram.Base.dll
   - Syncfusion.Diagram.Windows.dll
   - Syncfusion.Shared.Base.dll

3. Create a PaletteGroupBar control using the following code.

[C#]

```
            //Imports the Diagram control's namespace
            using Syncfusion.Windows.Forms.Diagram.Controls;

            //Creates a PaletteGroupBar instance
            PaletteGroupBar paletteBar = new PaletteGroupBar();
            paletteBar.Dock = DockStyle.Fill;
            paletteBar.Font = new Font("Arial", 9);
            paletteBar.BorderStyle = BorderStyle.None;

            //Apply visual styles
            paletteBar.VisualStyle =
Syncfusion.Windows.Forms.VisualStyle.Office2007;
            paletteBar.TextAlign =
Syncfusion.Windows.Forms.Tools.TextAlignment.Left;

            //Load palettes to paletteBar
            paletteBar.LoadPalette("..//..//Basic Shapes.edp");
            paletteBar.LoadPalette("..//..//Flowchart Symbols.edp");

            //Add paletteBar to the form
            this.Controls.Add(paletteBar);
```

[VB]

```
            'Imports the Diagram control's namespace
            Imports Syncfusion.Windows.Forms.Diagram.Controls

            'Creates a PaletteGroupBar instance
            Dim paletteBar As New PaletteGroupBar()
            paletteBar.Dock = DockStyle.Fill
            paletteBar.Font = New Font("Arial", 9)
            paletteBar.BorderStyle = BorderStyle.None
```

```
            'Apply visual styles
            paletteBar.VisualStyle =
Syncfusion.Windows.Forms.VisualStyle.Office2007
            paletteBar.TextAlign =
Syncfusion.Windows.Forms.Tools.TextAlignment.Left


            'Load palettes to paletteBar
            paletteBar.LoadPalette("..//..//Basic Shapes.edp")
            paletteBar.LoadPalette("..//..//Flowchart Symbols.edp")


            'Add paletteBar to the form
            Me.Controls.Add(paletteBar)
```

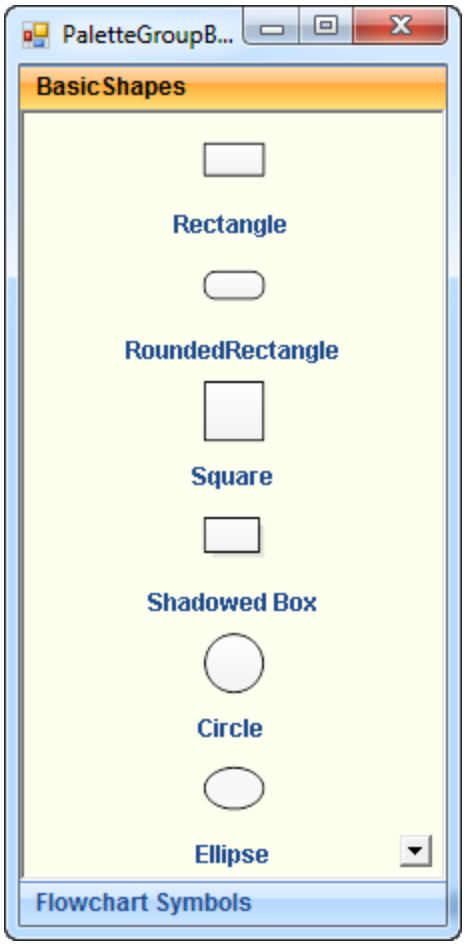

Figure 17: PaletteGroupBar Control Created using Code

3.2.3 PaletteGroupView

3.2.3.1 Creating a PaletteGroupView Control through Designer

This section depicts the step-by-step procedure to create a PaletteGroupView control through the Visual Studio designer in a .NET Windows Forms application.

To create a PaletteGroupView control using the designer:

1. Create a new Windows Forms application.
2. Open the **Designer Form** window.
3. Drag **PaletteGroupView** from **Toolbox** and drop it to the **Designer Form** window.

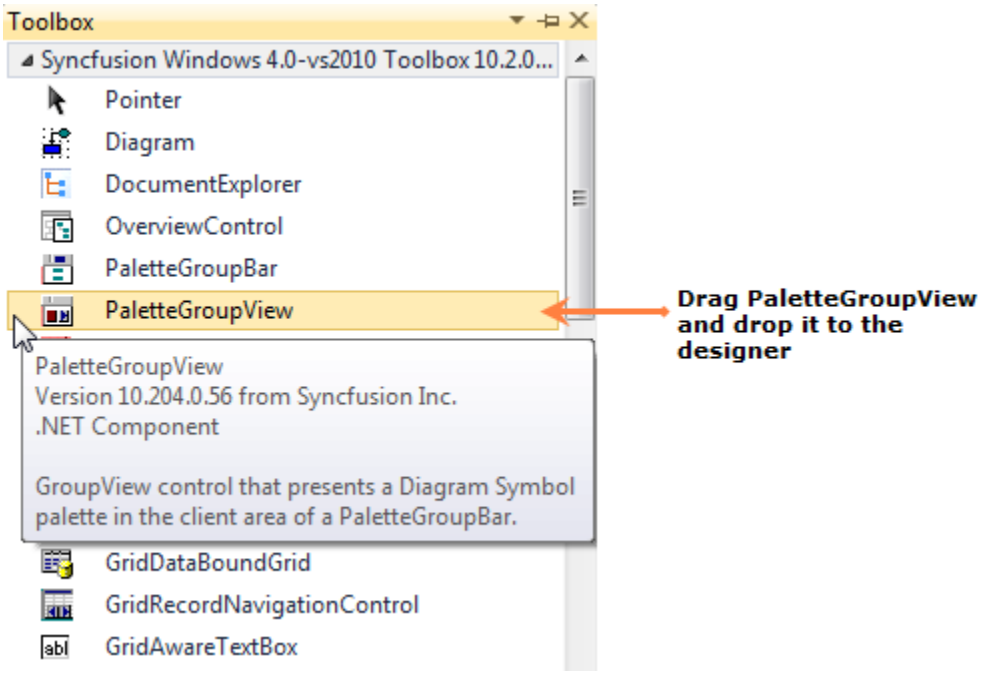

Figure 18: Dragging PaletteGroupView to the designer

The PaletteGroupView control will be added to the designer and its dependent assemblies will be added to the project once you dropped it to the Designer Form window.

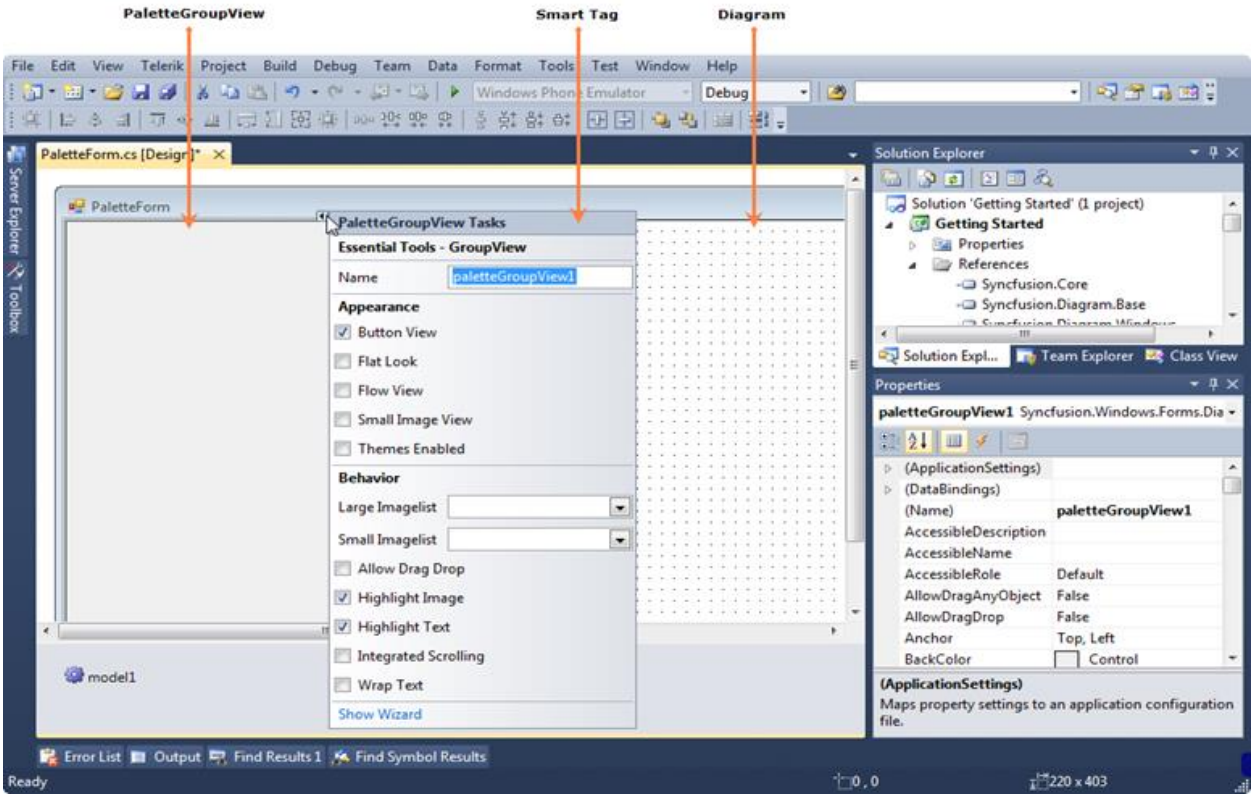

Figure 19: Designer Form Window Added with PaletteGroupView Control

3.2.3.2 Creating a PaletteGroupView Control through Code

This section shows the step-by-step procedure to create a PaletteGroupView control programmatically in a .NET Windows Form application.

To create a PaletteGroupView control using code:

1. Create a new Windows Form application.
2. Add the following basic dependent Syncfusion assemblies to the project:
   - Syncfusion.Core.dll
   - Syncfusion.Diagram.Base.dll
   - Syncfusion.Diagram.Windows.dll
   - Syncfusion.Shared.Base.dll

3. Create a PaletteGroupView control using the following code.

```
[C#]
            //Imports the Diagram control's namespace
            using Syncfusion.Windows.Forms.Diagram.Controls;
```

```
            //Creates a PaletteGroupView instance
            PaletteGroupView paletteView = new PaletteGroupView();

            //paletteView settings
            paletteView.Dock = DockStyle.Fill;
            paletteView.FlatLook = true;
            paletteView.BackColor = Color.White;
            paletteView.Font = new System.Drawing.Font("Arial", 9);

            //Load palette to paletteView
            paletteView.LoadPalette("..//..//Basic Shapes.edp");

            //Add the paletteView to the form
            this.Controls.Add(paletteView);
```

[VB]

```
            'Imports the Diagram control's namespace
            Imports Syncfusion.Windows.Forms.Diagram.Controls

            'Creates a PaletteGroupView instance
            Dim paletteView As New PaletteGroupView()

            'paletteView settings
            paletteView.Dock = DockStyle.Fill
            paletteView.FlatLook = True
            paletteView.BackColor = Color.White
            paletteView.Font = New System.Drawing.Font("Arial", 9)

            'Load palette to paletteView
            paletteView.LoadPalette("..//..//Basic Shapes.edp")

            'Add the paletteView to the form
```



```
            Me.Controls.Add(paletteView)
```

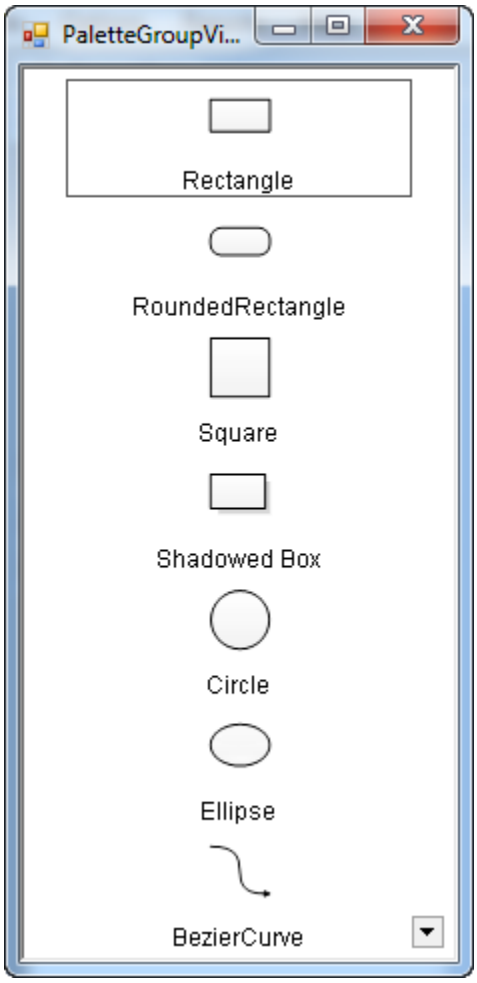

Figure 20: PaletteGroupView Control Created using Code

3.2.4 Overview Control

3.2.4.1 Creating an Overview Control through Designer

This section depicts the step-by-step procedure to create an Overview control through the Visual Studio designer in a .NET Windows Forms application.

To create an Overview control using the designer:

1. Create a new Windows Forms application.
2. Open the **Designer Form** window.
3. Drag **OverviewControl** from **Toolbox** and drop it to the **Designer Form** window.

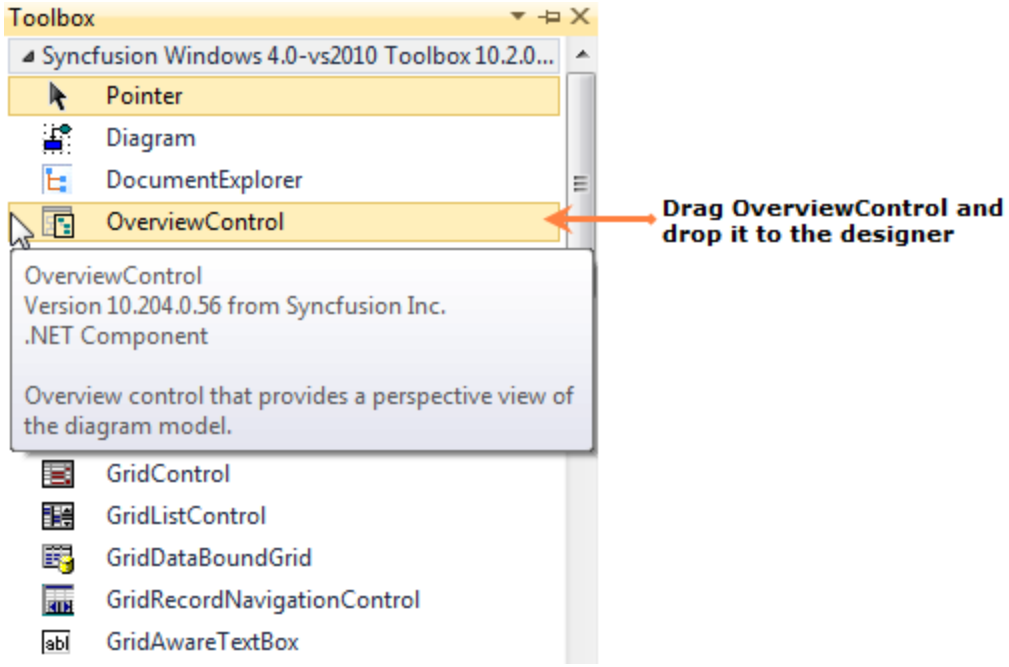

Figure 21: Dragging OverviewControl to the Designer

The Overview control will be added to the designer and its dependent assemblies will be added to the project once you dropped it to the Designer Form window.

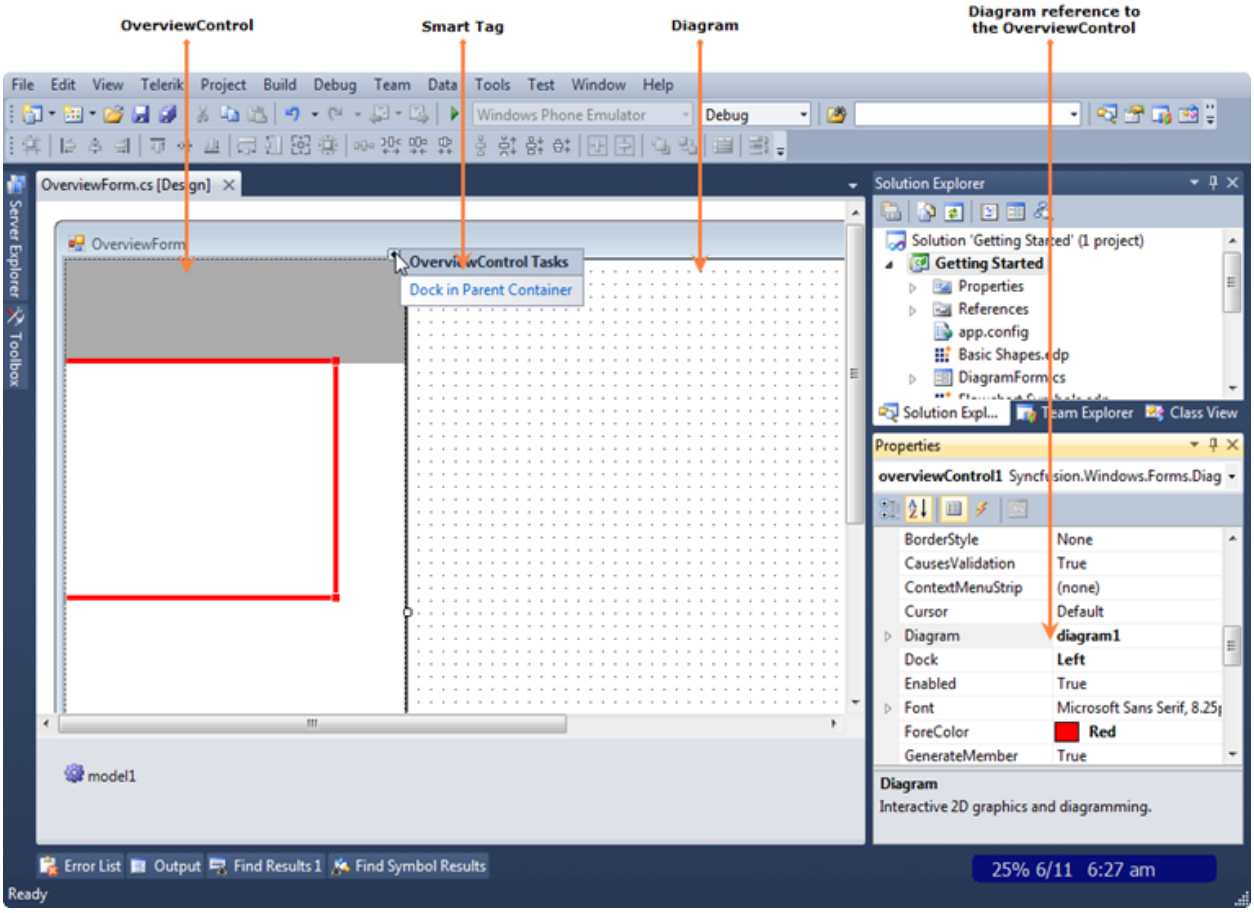

Figure 22: Designer Form Window Added with Overview Control

3.2.4.2 Creating an Overview Control through Code

This section shows the step-by-step procedure to create an Overview control programmatically in a .NET Windows Forms application.

To create an Overview control using code:

1. Create a new Windows Forms application.
2. Add the following basic dependent Syncfusion assemblies to the project:
   - Syncfusion.Core.dll
   - Syncfusion.Diagram.Base.dll
   - Syncfusion.Diagram.Windows.dll
   - Syncfusion.Shared.Base.dll

3. Create an Overview control using the following code.

[C#]

```
//Imports the Diagram control's namespace
using Syncfusion.Windows.Forms.Diagram.Controls;

//Creates an OverviewControl instance
OverviewControl overviewControl = new OverviewControl();
overviewControl.Dock = DockStyle.Left;

//Set the diagram reference to overviewControl
overviewControl.Diagram = diagram1;

//Add overviewControl to the form
this.Controls.Add(overviewControl);
```

[VB]

```
'Imports the Diagram control's namespace
Imports Syncfusion.Windows.Forms.Diagram.Controls

'Creates an OverviewControl instance
Dim overviewControl As New OverviewControl()
overviewControl.Dock = DockStyle.Left

'Set the diagram reference to overviewControl
overviewControl.Diagram = Diagram1

'Add overviewControl to the form
Me.Controls.Add(overviewControl)
```

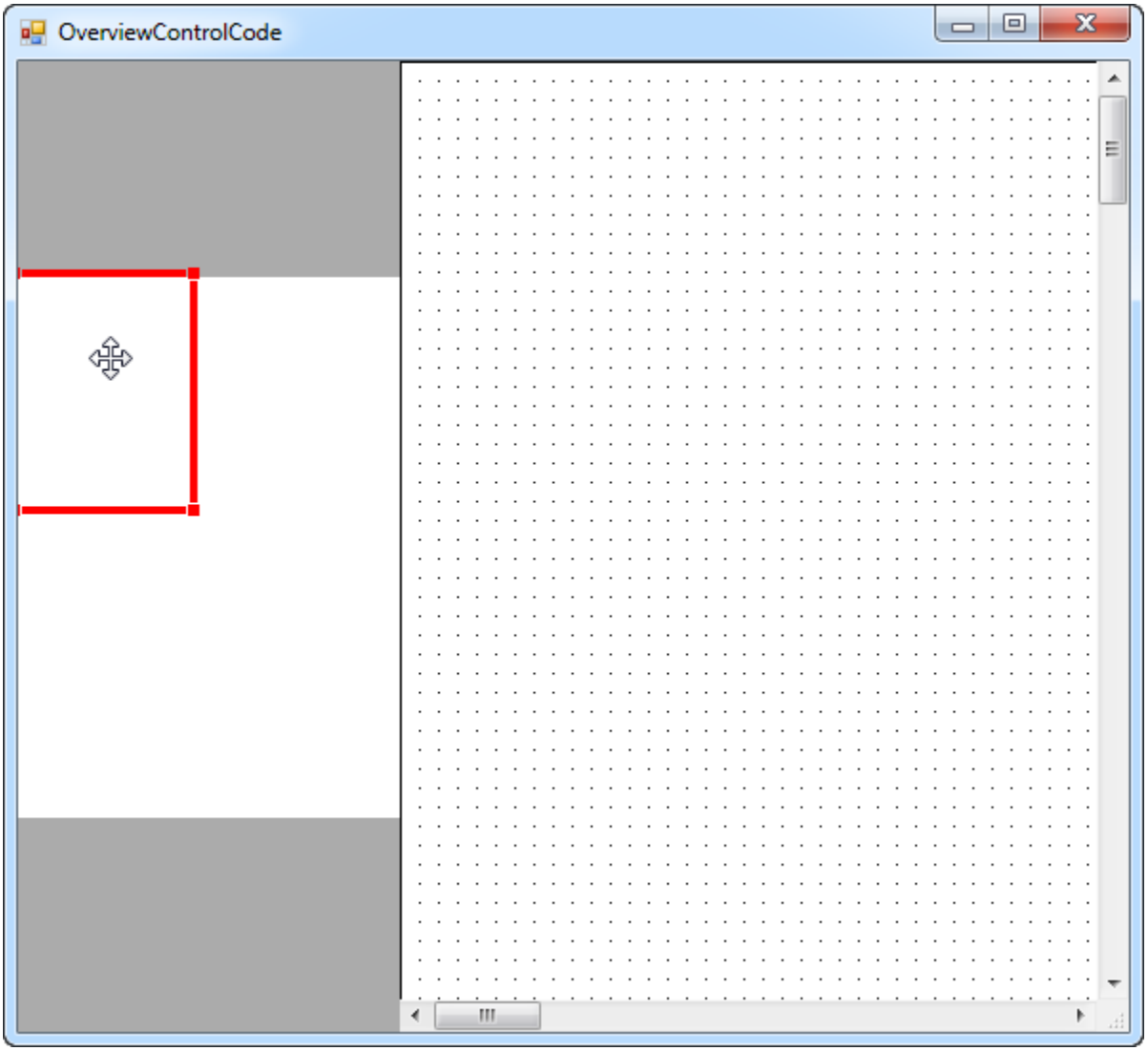

Figure 23: Overview Control Created using Code

3.2.5 PropertyEditor

3.2.5.1 Creating a PropertyEditor Control through Designer

This section depicts the step-by-step procedure to create a PropertyEditor control through the Visual Studio designer in a .NET Windows Forms application.

To create a PropertyEditor control using code:

1. Create a new Windows Forms application.
2. Open the **Designer Form** window.
3. Drag **PropertyEditor** from **Toolbox** and drop it to the **Designer Form** window.

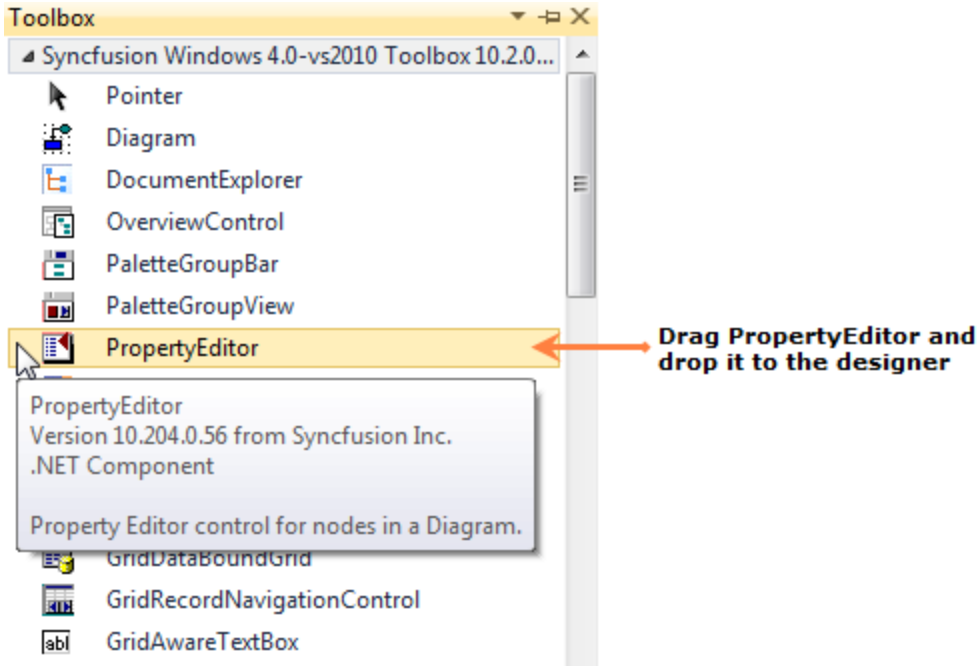

Figure 24: Dragging PropertyEditor to the Designer

The PropertyEditor control will be added to the designer and its dependent assemblies will be added to the project once you dropped it to the Designer Form window.

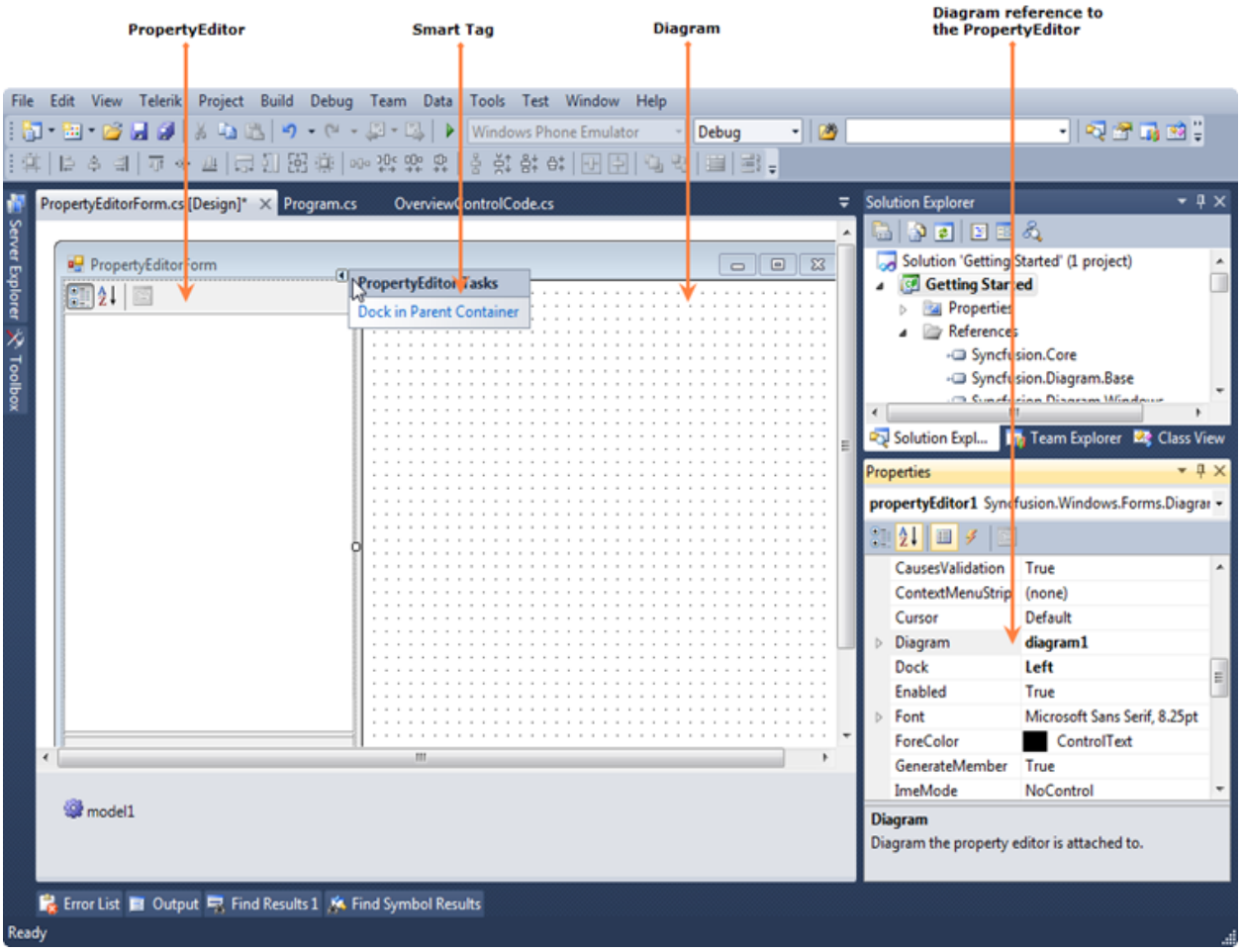

Figure 25: Designer Form Window with PropertyEditor Control

3.2.5.2 Creating a PropertyEditor Control through Code

This section shows the step-by-step procedure to create a PropertyEditor control programmatically in a .NET Windows Forms application.

To create a PropertyEditor control using code:

1. Create a new Windows Forms application.
2. Add the following basic dependent Syncfusion assemblies to the project:
   - Syncfusion.Core.dll
   - Syncfusion.Diagram.Base.dll
   - Syncfusion.Diagram.Windows.dll
   - Syncfusion.Shared.Base.dll

3. Create a PropertyEditor control using the following code.

[C#]

```
//Imports the Diagram control's namespace
using Syncfusion.Windows.Forms.Diagram.Controls;

//Creates a PropertyEditor instance
PropertyEditor propertyEditor = new PropertyEditor();
propertyEditor.Dock = DockStyle.Left;
propertyEditor.ShowCombo = true;

//Set the diagram reference to propertyEditor
propertyEditor.Diagram = diagram1;

//Add propertyEditor to the form
this.Controls.Add(propertyEditor);
```

[VB]

```
'Imports the Diagram control's namespace
Imports Syncfusion.Windows.Forms.Diagram.Controls

'Creates a PropertyEditor instance
Dim propertyEditor As New PropertyEditor()
propertyEditor.Dock = DockStyle.Left
propertyEditor.ShowCombo = True

'Set the diagram reference to propertyEditor
propertyEditor.Diagram = Diagram1

'Add propertyEditor to the form
Me.Controls.Add(propertyEditor)
```

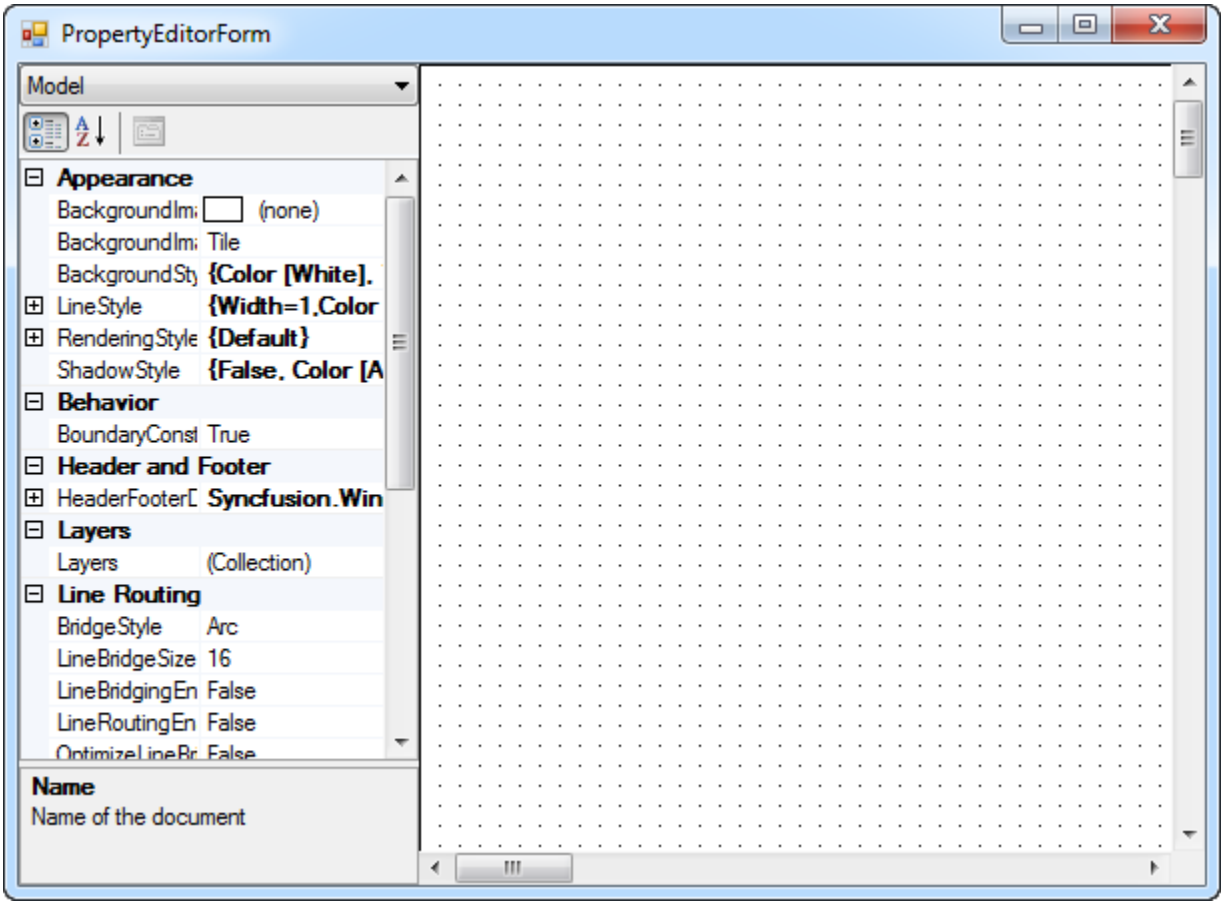

Figure 26: PropertyEditor Control Created using Code

3.2.6 DocumentExplorer

3.2.6.1 Creating a DocumentExplorer Control through Designer

This section depicts the step-by-step procedure to create a DocumentExplorer control through the Visual Studio designer in a .NET Windows Forms application.

To create a DocumentExplorer control using the designer:

1. Create a new Windows Forms application.
2. Open the **Designer Form** window.
3. Drag **DocumentExplorer** from **Toolbox** and drop it to the **Designer Form** window.

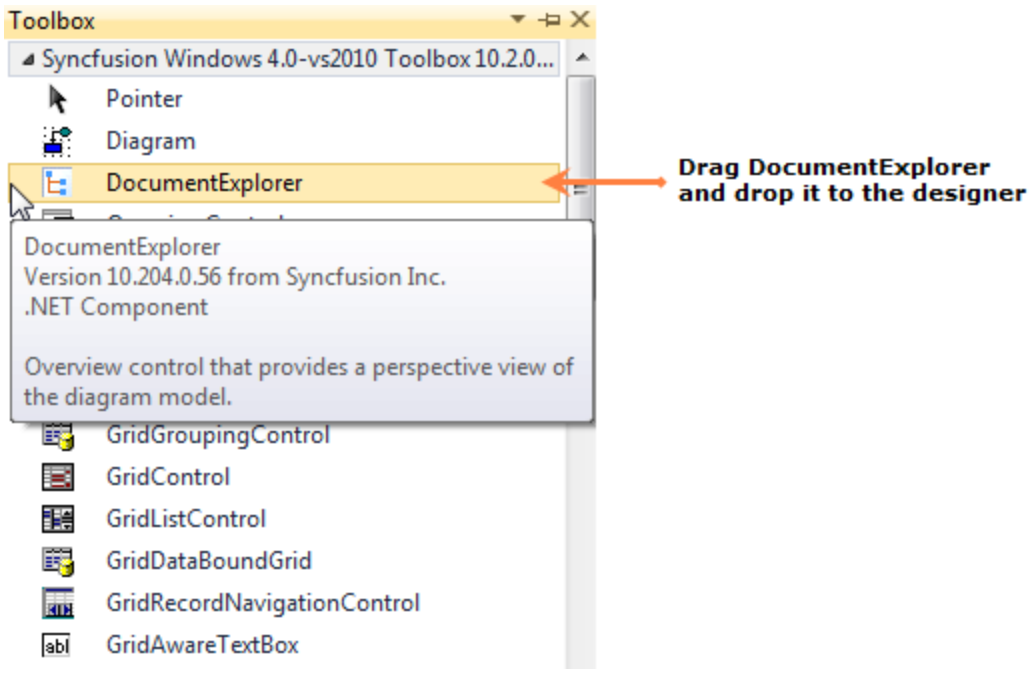

Figure 22: Dragging DocumentExplorer to the Designer

The DocumentExplorer control will be added to the designer and its dependent assemblies will be added to the project once you dropped it to the Designer Form window.

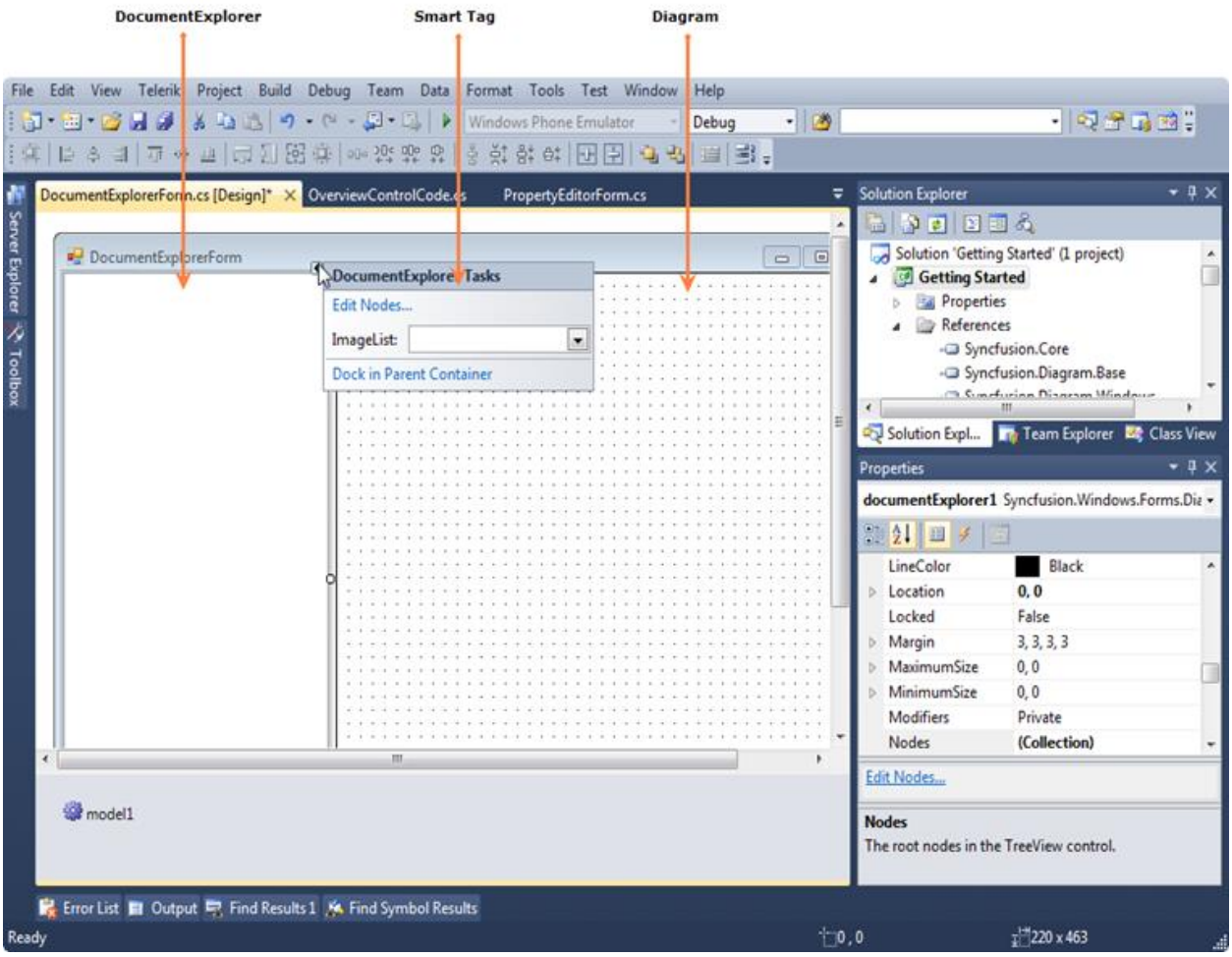

Figure 27: Designer Form Window Added with DocumentExplorer Control

3.2.6.2 Creating a DocumentExplorer Control through Code

This section shows the step-by-step procedure to create a DocumentExplorer control programmatically in a .NET Windows Forms application.

To create a DocumentExplorer control using code:

1. Create a new Windows Forms application.
2. Add the following basic dependent Syncfusion assemblies to the project:
   - Syncfusion.Core.dll
   - Syncfusion.Diagram.Base.dll
   - Syncfusion.Diagram.Windows.dll
   - Syncfusion.Shared.Base.dll

3. Create a DocumentExplorer control using the following code.

[C#]

```
//Imports the Diagram control's namespace
using Syncfusion.Windows.Forms.Diagram.Controls;

//Creates a DocumentExplorer instance
DocumentExplorer documentExplorer = new DocumentExplorer();
documentExplorer.Dock = DockStyle.Left;

//Attach a diagram model to documentExplorer
documentExplorer.AttachModel(diagram1.Model);

//Add documentExplorer to the form
this.Controls.Add(documentExplorer);
```

[VB]

```
'Imports the Diagram control's namespace
Imports Syncfusion.Windows.Forms.Diagram.Controls

'Creates a DocumentExplorer instance
Dim documentExplorer As New DocumentExplorer()
documentExplorer.Dock = DockStyle.Left

'Attach a diagram model to documentExplorer
documentExplorer.AttachModel(Diagram1.Model)

'Add documentExplorer to the form
Me.Controls.Add(documentExplorer)
```

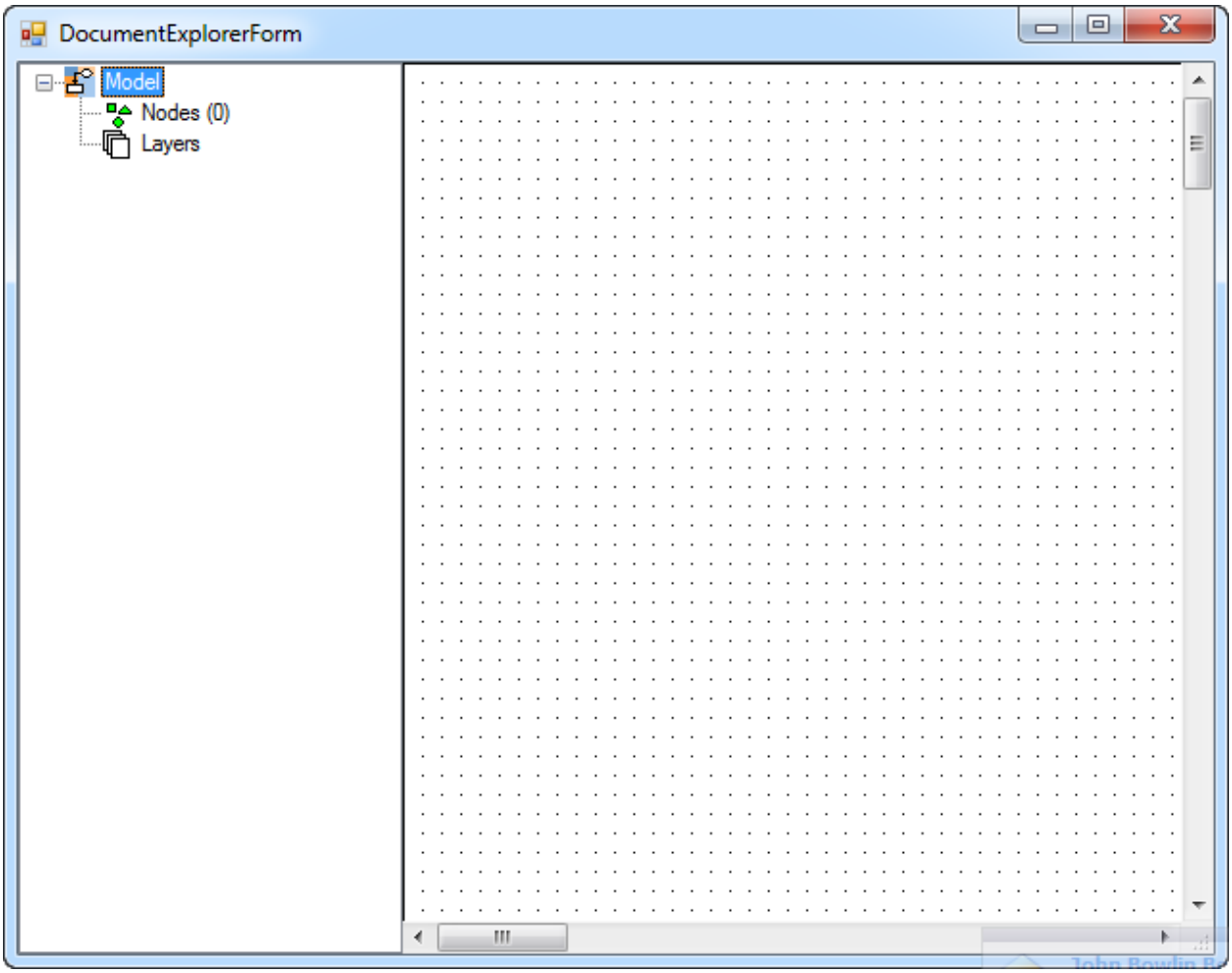

Figure 28: DocumentExplorer Control Created using Code

3.3 Diagram Builder

Diagram Builder application allows the user to create new diagrams and also modify the existing ones. This application has *.edd extension. The user can use this diagram in their applications.

The main difference between the diagram builder and symbol palette is as follows:

- In Diagram Builder, user creates diagram documents
- In Symbol Designer, user creates palettes.

Software Path

..\..\Syncfusion\Essential Studio\<Version Number>\utilities\Diagram\DiagramBuilder

1. Overview Control

Overview Control provides a perspective view of a diagram model, and allows users to dynamically pan and zoom the diagrams. The control features a view port window that can be moved and / or resized using the mouse to modify the diagrams' origin and magnification properties at run-time. The properties of this control is discussed in the Overview Control topic.

2. Palette GroupBar and GroupView

The **PaletteGroupBar** control provides a way for users to drag and drop symbols onto a diagram. It is based on the Syncfusion Essential Tools GroupBar control. Each symbol palette loaded in the PaletteGroupBar occupies a panel that can be selected by a bar button. The bar button is labeled with the name of the symbol palette. The symbols in the palette are shown as icons that can be dragged and dropped onto the diagram. This control allows users to add symbols to a palette, and save or load the palette whenever necessary. It provides a way to classify and maintain symbols.

The **PaletteGroupView** control provides an easy way to serialize a symbol palette to and from the resource file of a form. At design-time, users can attach a symbol palette to a PaletteGroupView control in the form. Selecting the PaletteGroupView and clicking the **Palette** property in the Visual Studio .NET Properties window will open a standard Open File dialog, which allows the user to select a symbol palette file that has been created with the Symbol Designer.

For more details about these diagram controls, refer to the Palette GroupBar and GroupView topic.

3. Property Editor

The Property Editor in Essential Diagram displays properties of the currently selected object(s) in the diagram. It is a Windows Forms control that can be added to the Visual Studio .NET Toolbox. It also allows users to set or modify various properties of the objects or the model. The Property Editor provides an easy interface to set and view the various property settings. To know about the control's properties see Property Editor topic.

4. Document Explorer

Document Explorer allows you to visualize the details of the various objects that are added onto the diagram control at run-time. The layers will be listed under the ***Layers*** node and other objects like shapes, links, lines and text editor will be listed under ***Nodes*** node.

5. Diagram Document

The DiagramDocument is a serializable document type that encapsulates the model and view data for the diagram. The grid area of the diagram document is the diagram view object area. The nodes dragged from the PaletteGroupBar will be dropped here.

For more details, see Diagram Grid topic.

Diagram Builder Functionalities

1. How to Open an Existing Diagram Document

Follow the below steps in order to open an existing diagram document

1. Add OpenFileDialog control to the Form.
2. Set the **Filter** property of OpenFileDialog as Essential Diagram Palettes|*.edp|Visio Stencils|*.vss; *.vsx|Visio Drawings(Shapes only)|*.vsd; *.vdx|All files|*.*.
3. Add the below code snippet in your button click event.

```
[C#]

// Checking whether "OK" button is clicked in OpenFileDialog
if (this.openFileDialog1.ShowDialog(this) == DialogResult.OK)
{
    string FileName = this.openFileDialog1.FileName;
    this.diagram1.LoadBinary(FileName);
    this.diagram1.Refresh();
}
```

The **diagram1.LoadBinary()** method loads the selected diagram file into diagram document.

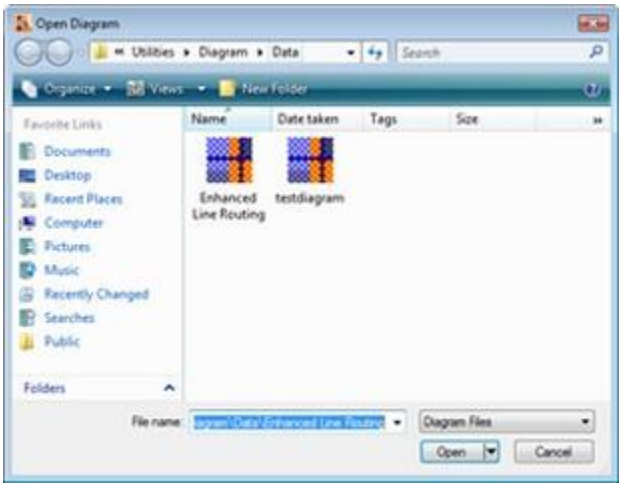

Figure 29: Diagram Open Dialog Box

2. How to Save a Diagram Document

Below are the steps to save a diagram document.

1. Add SaveFileDialog control to the Form.
2. Set the Filter property of SaveFileDialog as Essential Diagram Files|*.edd|All files|*.*.
3. Add the following code snippet in your button click event.

```
[C#]

// Checking whether "OK" button is clicked in SaveFileDialog
if (this.saveFileDialog1.ShowDialog(this) == DialogResult.OK)
{
    this.FileName = this.saveFileDialog1.FileName;
    this.diagram1.SaveBinary(this.FileName);
}
```

The **diagram1.SaveBinary()** method saves the diagram file in the given filename.

Figure 30: Diagram Save Dialog Box

3. How to print a Diagram Document

Following are the steps to print a diagram document:

1. Page Setup

The **Page Setup** dialog modifies the Page Settings and Printer Settings information for a given document. The user can enable sections of the dialog to manipulate printing, margins, paper orientation, size, source and to show help and network buttons. MinMargins defines the minimum margins a user can select.

The following code snippet can be used for setting the page set up for diagram document.

```
[C#]

if (diagram1 == null || diagram1.Model == null)
    return;
using (PageSetupDialog dlgPageSetup = new
        PageSetupDialog(diagram1.View))
{
    if (dlgPageSetup.ShowDialog() == DialogResult.OK)
    {
        diagram1.UpdateView();
    }
}
```

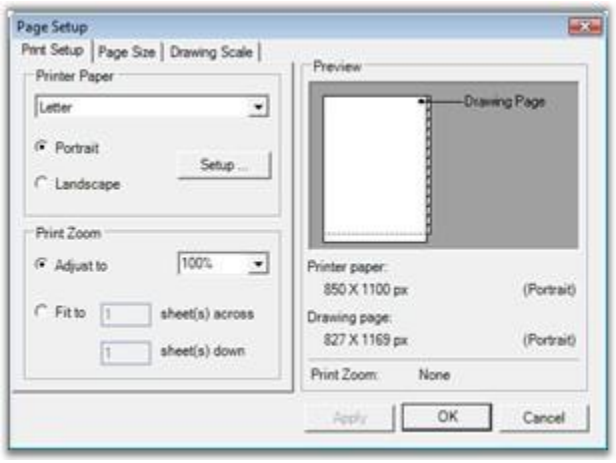

Figure 31: Diagram page Setup Dialog Box

2. Page Borders

The **Page Borders** dialog provides an interactive form-based interface, for setting the page borders of a diagram, initializing the dialog's Syncfusion.Windows.Forms.Diagram.PageBorderDialog. The **PageBorderStyle** property with the corresponding Syncfusion.Windows.Forms.Diagram.View.PageBorderStyle member of the diagram's view, will let the users to configure the page border settings using the dialog controls.

The following code snippet can be used for setting the page border for diagram document.

```
[C#]

if (diagram1 != null && diagram1.Model != null)
```



```
{
    PageBorderDialog borderDialog = new PageBorderDialog();
    borderDialog.PageBorderStyle = diagram1.View.PageBorderStyle; // It
will show existing border set up
    if (borderDialog.ShowDialog() == DialogResult.OK)
    {
        diagram1.View.PageBorderStyle = borderDialog.PageBorderStyle;
// It will update the modified set up.
        diagram1.View.RefreshPageSettings();
        diagram1.UpdateView();
    }
}
```

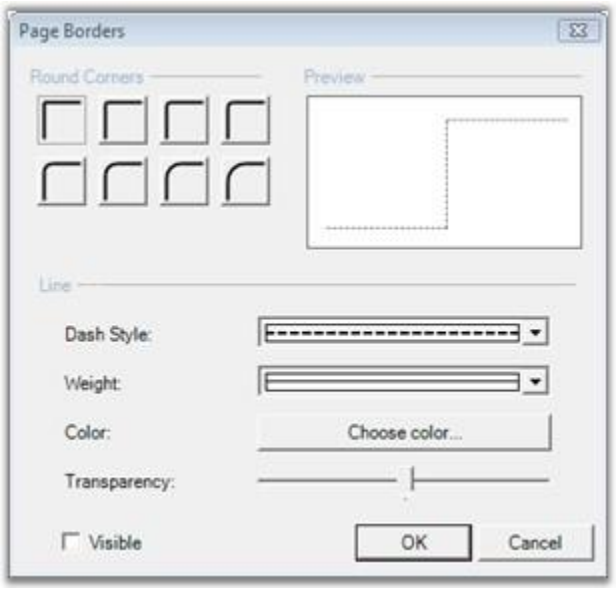

Figure 32: Diagram Page Borders Dialog Box

3. Header and Footers

The **Header and Footer** dialog provides an interactive form-based interface for initializing the Header and Footer settings of a diagram.

The following code snippet can be used for creating the Header and Footer dialog.

```
[C#]

if (diagram1 != null && diagram1.Model != null)
{
    HeaderFooterDialog dlgHF = new HeaderFooterDialog();
```



```
    dlgHF.Header = diagram1.Model.HeaderFooterData.Header;
    dlgHF.Footer = diagram1.Model.HeaderFooterData.Footer;
    dlgHF.MeasurementUnits = diagram1.Model.MeasurementUnits;
    if (dlgHF.ShowDialog() == DialogResult.OK)
    {
        diagram1.Model.HeaderFooterData.Header = dlgHF.Header;
        diagram1.Model.HeaderFooterData.Footer = dlgHF.Footer;
    }
}
```

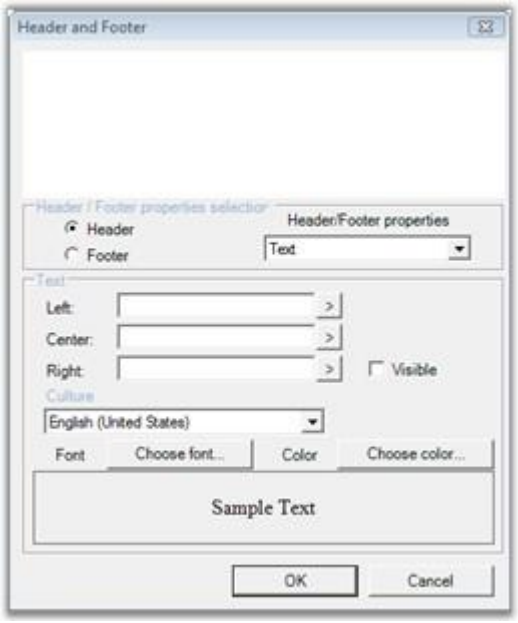

Figure 33: Diagram Header Footer Dialog Box

4. Print Preview

It will show a preview of the page which will appear when printed. The **Print Preview** dialog shows the preview of the page with the following:

- Page setup
- Page border set up
- Header and footers in the page

The following code snippet can be used for creating Print Preview dialog.

```
[C#]

if (diagram1 != null)
```

```
{
    PrintDocument printDoc = diagram1.CreatePrintDocument();
    PrintPreviewDialog printPreviewDlg = new PrintPreviewDialog();
    printPreviewDlg.StartPosition = FormStartPosition.CenterScreen;

    printDoc.PrinterSettings.FromPage = 0;
    printDoc.PrinterSettings.ToPage = 0;
    printDoc.PrinterSettings.PrintRange = PrintRange.AllPages;

    printPreviewDlg.Document = printDoc;
    printPreviewDlg.ShowDialog(this);
}
```

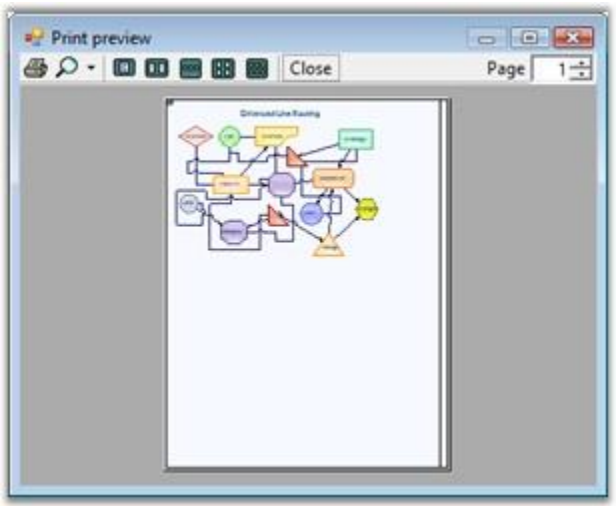

Figure 34: Print Preview Dialog Box

5. Print

This option will send the diagram document to the printer.

The following code snippet can be used for sending the document for printing.

```
[C#]

if (diagram1 != null)
{
    PrintDocument printDoc = diagram1.CreatePrintDocument();
    PrintDialog printDlg = new PrintDialog();
    printDlg.Document = printDoc;
```

```
    printDlg.AllowSomePages = true;

    if (printDlg.ShowDialog(this) == DialogResult.OK)
    {
        printDoc.PrinterSettings = printDlg.PrinterSettings;
        printDoc.Print();
    }
}
```

Diagram Builder Tools

Editing Options

Edit Menu Items	Description	Code Snippet
Undo	Reverts the latest modification done.	`Diagram1.Model.HistoryManager.Undo();`
Redo	Steps forward to operation history records and redoes the last undone task.	`Diagram1.Model.HistoryManager.Redo();`
Cut	Removes the currently selected nodes from the diagram and move them to the clipboard.	`Diagram1.Controller.Cut();`
Copy	Copies the currently selected nodes to the clipboard.	`Diagram1.Controller.Copy();`
Paste	Pastes the contents of the clipboard to the diagram.	`Diagram1.Controller.Paste();`
Select All	Adds all nodes in the diagram model to the SelectionList.	`Diagram1.Controller.SelectAll();`

Pan & Zoom Tool

The following screen shot illustrates the pan and zoom tools.

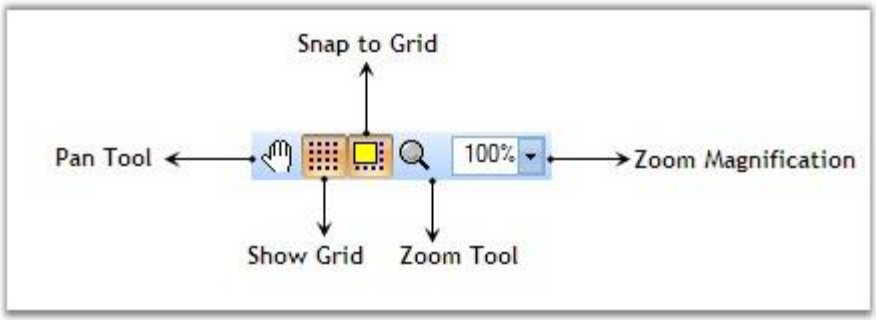

Figure 35: Pan&Zoom Tool

Tool Name	Description	Code Snippet
Pan Tool	Pan tool allows the user to drag the diagram and hence scroll it in any direction.	`diagram1.Controller.ActivateTool("PanTool");`
Zoom Tool	Zoom tool allows the user to zoom the diagram with minimum and maximum magnification.	`diagram1.Controller.ActivateTool("ZoomTool");`
Magnification	This value is used to zoom the view in and out. The x and y axes can be scaled independently. Normally, the x and y axes will have the same magnification value.	`int magVal = 30;` `diagram1.View.Magnification= magVal;`
ShowGrid	This will show / hide the diagram view grid.	`Diagram1.View.Grid.Visible = true;`
SnapToGrid	Specifies whether the snap to grid feature is enabled.	`Diagram1.View.Grid.SnapToGrid =true;`
Rulers	Diagram control supports rulers similar to that in Microsoft Word. For details see Rulers	`Diagram1.ShowRulers=true;`

Alignment Tool

The following screen shot illustrates the Alignment tools.

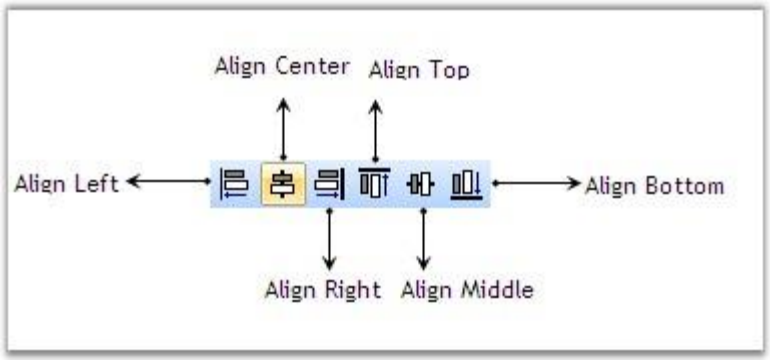

Figure 36: Alignment Tool

Tool Name	Description	Code Snippet
AlignLeft	Aligns the selected nodes along the left edge of the first node.	`diagram1.AlignLeft();`
AlignCenter	Aligns the selected nodes along the vertical center of the first node.	`diagram1.AlignCenter();`
AlignRight	Aligns the selected nodes along the right edge of the first node.	`diagram1.AlignRight();`
AlignTop	Aligns the selected nodes along the top edge of the first node.	`diagram1.AlignTop();`
AlignMiddle	Aligns the selected nodes along the horizontal center of the first node.	`diagram1.AlignMiddle();`
AlignBottom	Aligns the selected nodes along the bottom edge of the first node.	`diagram1.AlignBottom();`

Rotate Tool

The following screen shot illustrates the Rotate tools.

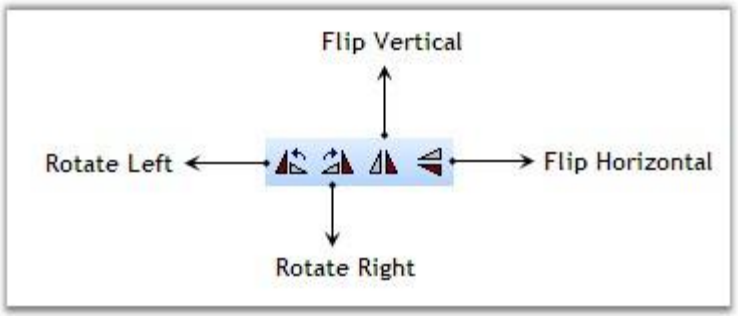

Figure 37: Rotate Tools

Tool Name	Description	Code Snippet
RotateLeft	Rotates the selected nodes about their local origin by -90 degrees.	`diagram1.Rotate(-90);`
RotateRight	Rotates the selected nodes about their local origin by 90 degrees.	`diagram1.Rotate(90);`
FlipVertical	Flips the selected nodes about their vertical (Y) axis.	`diagram1.FlipVertical();`
FlipHorizontal	Flips the selected nodes about their horizontal (X) axis.	`diagram1.FlipHorizontal();`

Resize Tool

The following screen shot illustrates the Resize tools.

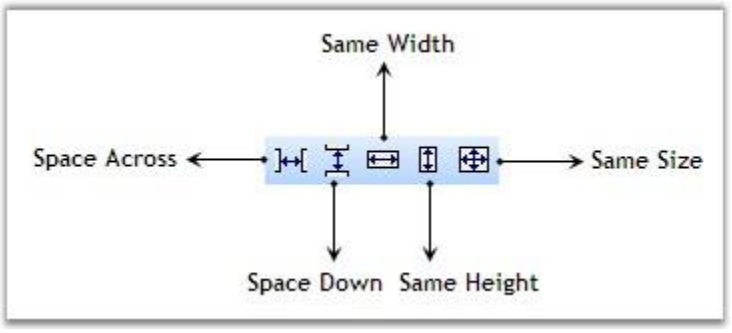

Figure 38: Resize Tools

Tool Name	Description	Code Snippet

SpaceAcross	Positions the selected nodes for equal horizontal spacing	`diagram1.SpaceAccross();`
SpaceDown	Positions the selected nodes for equal vertical spacing	`diagram1.SpaceDown();`
SameSize	Sets the width and height of the selected nodes to be equal.	`diagram1.SameSize();`
SameHeight	Sets the height of the selected nodes to be equal.	`diagram1.SameHeight();`
SameWidth	Sets the width of the selected nodes to be equal.	`diagram1.SameWidth();`

Nudge Tool

The following screen shot illustrates the Nudge tools.

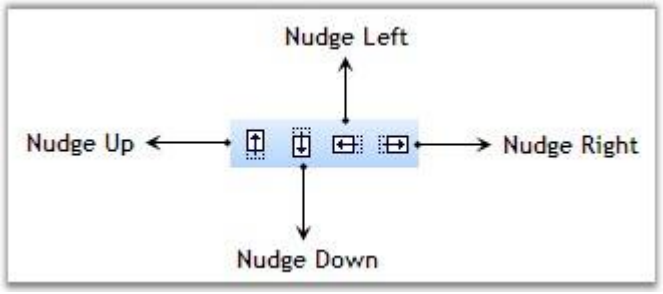

Figure 39: Nudge Tools

Tool Name	Description	Code Snippet
NudgeUp	Nudge the selected components up by Syncfusion.Windows.Forms.Diagram.Controls.Diagram.NudgeIncrement units.	`diagram1.NudgeUp();`
NudgeDown	Nudge the selected components down by Syncfusion.Windows.Forms.Diagram.Controls.Diagram.NudgeIncrement units.	`diagram1.NudgeDown();`
NudgeLeft	Nudge the selected components to the left by Syncfusion.Windows.Forms.Diagram.Controls.Diagram.NudgeIncrement units.	`diagram1.NudgeLeft();`

NudgeRight	Nudge the selected components to the right by Syncfusion.Windows.Forms.Diagram.Controls.Diagram.NudgeIncrement units.	`diagram1.NudgeRight();`

Text Formatting Tool

The following screen shot illustrates the Text Formatting tools.

Figure 40: Text Formatting Tools

Tool Name	Description	Code Snippet

Font Family	The **FamilyName** property is used to get or set the font family name.	`string strFamilyName = this.comboBoxBarItemFontFamily.ListBox.SelectedItem.ToString();`
		
		`if(this.diagram1.Controller.TextEditor.FamilyName != strFamilyName )this.diagram1.Controller.TextEditor.FamilyName = strFamilyName;`

Font Size	Gets or sets the size of the point.	`int ptSize = 10;` `this.diagram1.Controller.TextEditor.PointSize = ptSize;`
Bold	Gets or sets a value indicating whether the Syncfusion.Windows.Forms.Diagram.TextEditor is bold.	`bool newValue = !( this.diagram1.Controller.TextEditor.Bold );` `this.diagram1.Controller.TextEditor.Bold = newValue;`
Italic	Gets or sets a value indicating whether the Syncfusion.Windows.Forms.Diagram.TextEditor is italic.	`bool newValue = !( this.diagram1.Controller.TextEditor.Italic );` `this.diagram1.Controller.TextEditor.Italic = newValue;`
Underline	Gets or sets a value indicating whether the Syncfusion.Windows.Forms.Diagram.TextEditor is underline.	`bool newValue = !( this.diagram1.Controller.TextEditor.Underline );` `this.diagram1.Controller.TextEditor.Underline = newValue;`
StrikeOut	Gets or sets a value indicating whether	`bool newValue = !( this.diagram1.Controller.TextEdi`

	the Syncfusion.Windows.Forms.Diagram.TextEditor is strikeout.	tor.Strikeout); this.diagram1.Controller.TextEditor.Strikeout = newValue;

TextColor	Gets or sets the color of the text.	ColorDialog dlg = new ColorDialog( );
		dlg.Color
		this.diagram1.Controller.TextEditor.TextColor;
		if (dlg.ShowDialog(this) == DialogResult.OK)
		{
		this.diagram1.Controller.TextEditor.TextColor = dlg.Color;
		}

Align Text Left	Gets or sets the horizontal alignment to Near.	this.diagram1.Controller.TextEditor.HorizontalAlignment = StringAlignment.Near;
Align Text Right	Gets or sets the horizontal alignment to Far.	this.diagram1.Controller.TextEditor.HorizontalAlignment= StringAlignment.Far;
Align Text Center	Gets or sets the horizontal alignment to Center	this.diagram1.Controller.TextEditor.HorizontalAlignment = StringAlignment.Center;
Subscript	Gets or sets a value indicating whether the Syncfusion.Windows.Forms.Diagram.TextEditor is subscript.	bool newValue = !(this.diagram1.Controller.TextEditor.Subscript); this.diagram1.Controller.TextEditor.Subscript = newValue;
Superscript	Gets or sets a value indicating whether the Syncfusion.Windows.Forms.Diagram.TextEditor is superscript.	bool nValue = !(this.diagramComponent.Controller.TextEditor.Superscript); this.diagramComponent.Controller.TextEditor.Superscript = newValue;

Lower Text	Decreases the char offset value.	int nCurrentOffset = this.diagramComponent.Controller.TextEditor.CharOffset;
		nCurrentOffset--;
		this.diagram1.Controller.TextEditor.CharOffset = nCurrentOffset;

Upper Text	Increases the char offset value.	int nCurrentOffset = this.diagram1.Controller.TextEditor.CharOffset;
		nCurrentOffset++;
		this.diagram1.Controller.TextEdi

		tor.CharOffset = nCurrentOffset;

Group & Order Tool

The following screen shot illustrates the Group and Order tools.

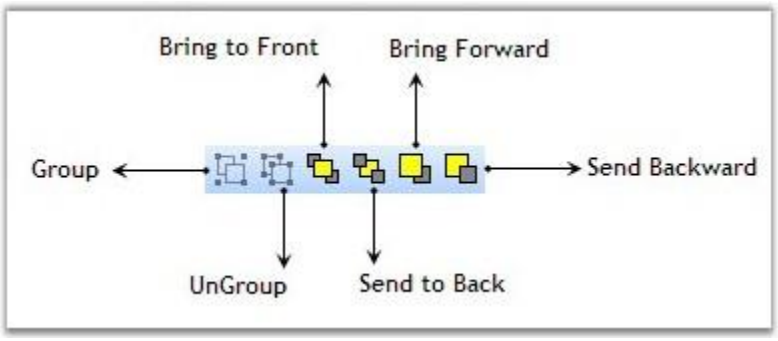

Figure 41: Group & Order Tools

Tool Name	Description	Code Snippet
Group	Groups the currently selected nodes in a diagram Group.	`diagram1.Controller.Group();`
UnGroup	Ungroups the currently selected group in a diagram.	`diagram1.Controller.UnGroup();`
BringToFront	Brings the selected nodes to the front of the Z-order.	`diagram1.Controller.BringToFront();`
SendToBack	Sends the selected nodes to the back of the Z-order.	`diagram1.Controller.SendToBack();`
BringForward	Brings the selected nodes forward in the Z-order.	`Diagram1.Controller.BringForward();`
SendBackward	Sends the selected nodes backward in the Z-order.	`Diagram1.Controller.SendBackward();`

Drawing Tools

The following screen shot illustrates the drawing tools.

Figure 42: Drawing Tools

Tool Name	Description	Code Snippet
SelectTool	Specifies the selection mode.	`diagram1.Controller.ActivateTool("SelectTool");`
LineTool	Draws straight line with start and end point.	`diagram1.Controller.ActivateTool("LineTool");`
PolyLineTool	Interactive tool for drawing polylines.	`diagram1.Controller.ActivateTool("PolyLineTool");`
RectangleTool	Interactive tool for drawing rectangles.	`diagram1.Controller.ActivateTool("RectangleTool");`
RectangleTool	Interactive tool for drawing rounded rectangles.	`diagram1.Controller.ActivateTool("RectangleTool");`
EllipseTool	Interactive tool for drawing ellipses.	`diagram1.Controller.ActivateTool("EllipseTool");`
PolygonTool	Interactive tool for drawing polygons.	`diagram1.Controller.ActivateTool("PolygonTool");`
CurveTool	Interactive tool for drawing curves.	`diagram1.Controller.ActivateTool("CurveTool");`
ClosedCurveTool	Interactive tool for drawing closed curves.	`diagram1.Controller.ActivateTool("ClosedCurveTool");`
PencilTool	Draws the user defined shape similar to Microsoft Paint.	`diagram1.Controller.ActivateTool("PencilTool");`
SplineTool	Interactive tool for drawing spline.	`diagram1.Controller.ActivateTool("SplineTool");`
BezierTool	Interactive tool for drawing bezier.	`diagram1.Controller.ActivateTool("BezierTool");`
TextTool	Interactive tool for inserting text nodes into a diagram and editing	`diagram1.Controller.ActivateTool("TextTool");`

	existing text nodes. This tool manages the insertion of new text nodes into a diagram and editing existing ones. Activating this tool causes it to track mouse-down, mouse-move, and mouse-up events and draw a tracking rectangle. The rectangle drawn is used as the bounds of a new text node, which is inserted into the diagram using an InsertNodesCmd. This tool also listens to the double-click events. If the user double-clicks a text node, this tool opens a text editor allowing the user to edit the text.	
RichTextTool	Interactive tool for inserting and editing rich text objects. This tool manages the insertion of new rich text nodes into a diagram and editing of existing rich text nodes. Activating this tool causes it to track mouse-down, mouse-move, and mouse-up events and draw a tracking rectangle. The rectangle drawn is used as the bounds of a new rich text node, which is inserted into the diagram using an InsertNodesCmd command. This tool also listens to the	`diagram1.Controller.ActivateTool("RichTextTool");`

	double-click events. If the user double-clicks a rich text node, this tool opens a text editor allowing the user to edit the text.	
BitmapTool	Interactive tool for inserting bitmaps into a diagram.	`diagram1.Controller.ActivateTool("BitmapTool");`
ConnectionPointTool	The connection point tool is an interactive tool for inserting and deleting connection points on diagram nodes. You can insert a connection point by clicking the node and delete a connection point by holding CTRL and clicking the node.	diagram1.Controller.ActivateTool("ConnectionPointTool");

Diagram Connector Tools

The following screen shot illustrates the Diagram Connector tools.

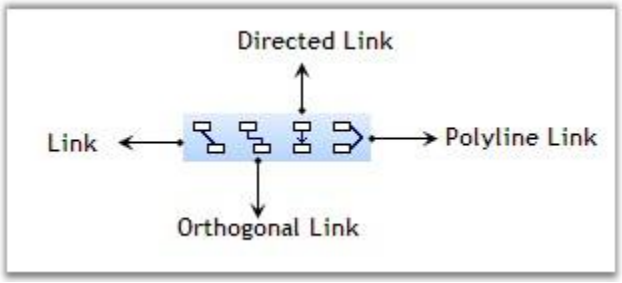

Figure 43: Diagram Connector Tools

LineConnectorTool

Line Connector Tool is used to connect nodes in a straight line. It creates line shape nodes. The name of the LineConnectorTool is **LineLinkTool**.

The below table lists the properties of the tool.

Property	Description
HeadDecorator	Sets the Head Decorator applied to the created node.

TailDecorator	Sets the Tail Decorator applied to the created node.
InAction	Sets the distance from the start of the line to the dash pattern. It accepts Float value.
Name	Sets the Name for the Tool.
Preceding Tool	Gets the Preceding Tool.

```
[C#]

diagram1.Controller.ActivateTool("LineLinkTool");

Tool t = diagram1.Controller.ActiveTool;
if (t is Syncfusion.Windows.Forms.Diagram.LineConnectorTool)
{
    LineConnectorTool l = (LineConnectorTool)t;
    l.HeadDecorator.DecoratorShape = DecoratorShape.Filled45Arrow;
    l.TailDecorator.DecoratorShape = DecoratorShape.Filled45Arrow;
}
```

Orthogonal Connector Tool

Orthogonal Connector Tool is used to connect nodes in an orthogonal manner by providing its start point and end point. It creates the Orthogonal Line Shape node. The name of the Orthogonal Connector Tool is **OrthogonalLinkTool**. The below table lists the properties of the tool.

Property	Description
HeadDecorator	Sets the Head Decorator applied to the created node.
TailDecorator	Sets the Tail Decorator applied to the created node.
InAction	Sets the distance from the start of the line to the dash pattern. It accepts Float value.
Name	Sets the Name for the Tool.
Preceding Tool	Gets the Preceding Tool.

```
[C#]
```

```
diagram1.Controller.ActivateTool("OrthogonalLinkTool");
Tool t = diagram1.Controller.ActiveTool;
if (t is Syncfusion.Windows.Forms.Diagram.OrthogonalConnectorTool)
{
    OrthogonalConnectorTool l = (OrthogonalConnectorTool)t;
    l.HeadDecorator.DecoratorShape = DecoratorShape.Filled45Arrow;
    l.TailDecorator.DecoratorShape = DecoratorShape.Filled45Arrow;
}
```

DirectedLineConnector Tool

DirectedLineConnector Tool is used to connect the nodes in a directed line. It creates the directed line shape node. The name of the DirectedLineConnectorTool is **DirectedLineLinkTool**. The below table lists the properties of the tool.

Property	Description
HeadDecorator	Sets the Head Decorator applied to the created node.
TailDecorator	Sets the Tail Decorator applied to the created node.
InAction	Sets the distance from the start of the line to the dash pattern. It accepts Float value.
Name	Sets the Name for the Tool.
Preceding Tool	Gets the Preceding Tool.

```
[C#]

diagram1.Controller.ActivateTool("DirectedLineLinkTool");
Tool t = diagram1.Controller.ActiveTool;
if (t is Syncfusion.Windows.Forms.Diagram.DirectedLineConnectorTool)
{
    DirectedLineConnectorTool l = (DirectedLineConnectorTool)t;
    l.HeadDecorator.DecoratorShape = DecoratorShape.Filled45Arrow;
    l.TailDecorator.DecoratorShape = DecoratorShape.Filled45Arrow;
}
```

PolyLineConnector Tool

This is an interactive tool for drawing Polyline Connector. The name of the tool is "PolyLineLinkTool". The below table lists the properties of the PolyLine tool.

Property	Description
HeadDecorator	Sets the Head Decorator applied to the created node.
TailDecorator	Sets the Tail Decorator applied to the created node.
InAction	Sets the distance from the start of the line to the dash pattern. It accepts Float value.
Name	Sets the Name for the Tool.
Preceding Tool	Gets the Preceding Tool.

```
[C#]

diagram1.Controller.ActivateTool("PolyLineLinkTool");
Tool t = diagram1.Controller.ActiveTool;
if (t is Syncfusion.Windows.Forms.Diagram.PolyLineConnectorTool)
{
    PolyLineConnectorTool l = (PolyLineConnectorTool)t;
    l.HeadDecorator.DecoratorShape = DecoratorShape.Filled45Arrow;
    l.TailDecorator.DecoratorShape = DecoratorShape.Filled45Arrow;
}
```

Creating a Diagram using Diagram Builder

To create your own diagram in the diagram builder, follow the below given procedure.

1. Go to the **File** menu and click **New**. The new window is displayed as in the following screen shot.

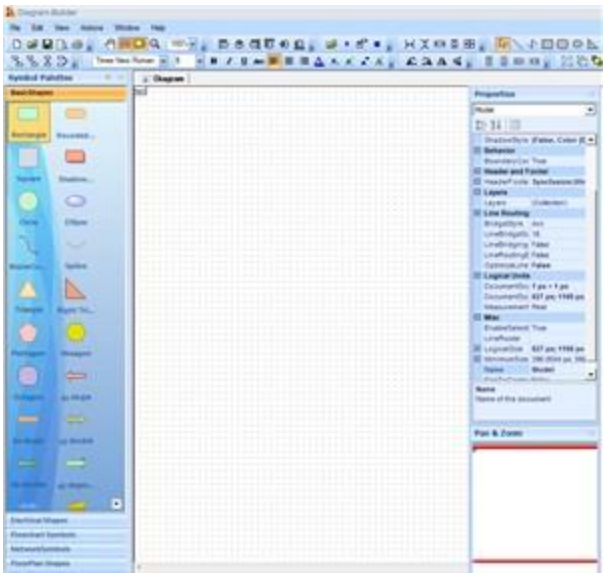

Figure 44: Diagram Builder

2. To add symbols into the symbol palette, select **Add SymbolPalette** in the **File** menu.

3. Select the symbol palette, which you created previously using symbol designer from the list of Symbol Palettes displayed.

Figure 45: Symbol Palette

4. On placing the symbol into the diagram area, the Diagram Builder displays a dialog for adding the symbol palette into the Associated Palettes. To add, click **OK**. Click **Cancel** if not required.

5. Place the symbols and change their properties according to the requirements. Finally save the file with .edd extension.

A diagram is created using the Diagram Builder. You can use this diagram (.edd) file for developing your application.

Figure 46: Sample Diagram

3.4 Symbol Designer

Symbol Designer application allows you to create new palettes with symbols, and also modify the existing palettes. These palettes have the *.edp extension. You can use these palettes in your applications, and also in the Diagram Builder.

Software Path

..\..\Syncfusion\Essential Studio\<Version Number>\utilities\Diagram\Symbol Designer

Creating EDP File

To create our own custom symbols in the symbol designer, follow the procedure given below.

- Open the Symbol Designer tool which is available in the above software path.
- Symbol designer contains below three sections,

a. Symbol palette
b. Object model
c. Properties Window

- If you want to create a new palette, select **New** option in the **File** menu. Ttype a name for the palette as shown in the below sample and click **OK**.

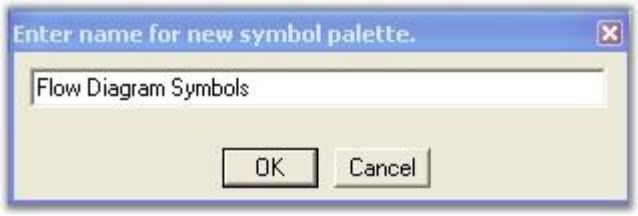

Figure 47: New Symbol Palette Dialog Box

- A new symbol palette is created with the given name. Now, you can add your own symbols into this palette.

Figure 48: Flow Diagram Symbol Palette

- For creating a new symbol, click **Add Symbol** option in the **Symbol** menu. The symbol designer window changes to a new format as shown in the below screen shot.

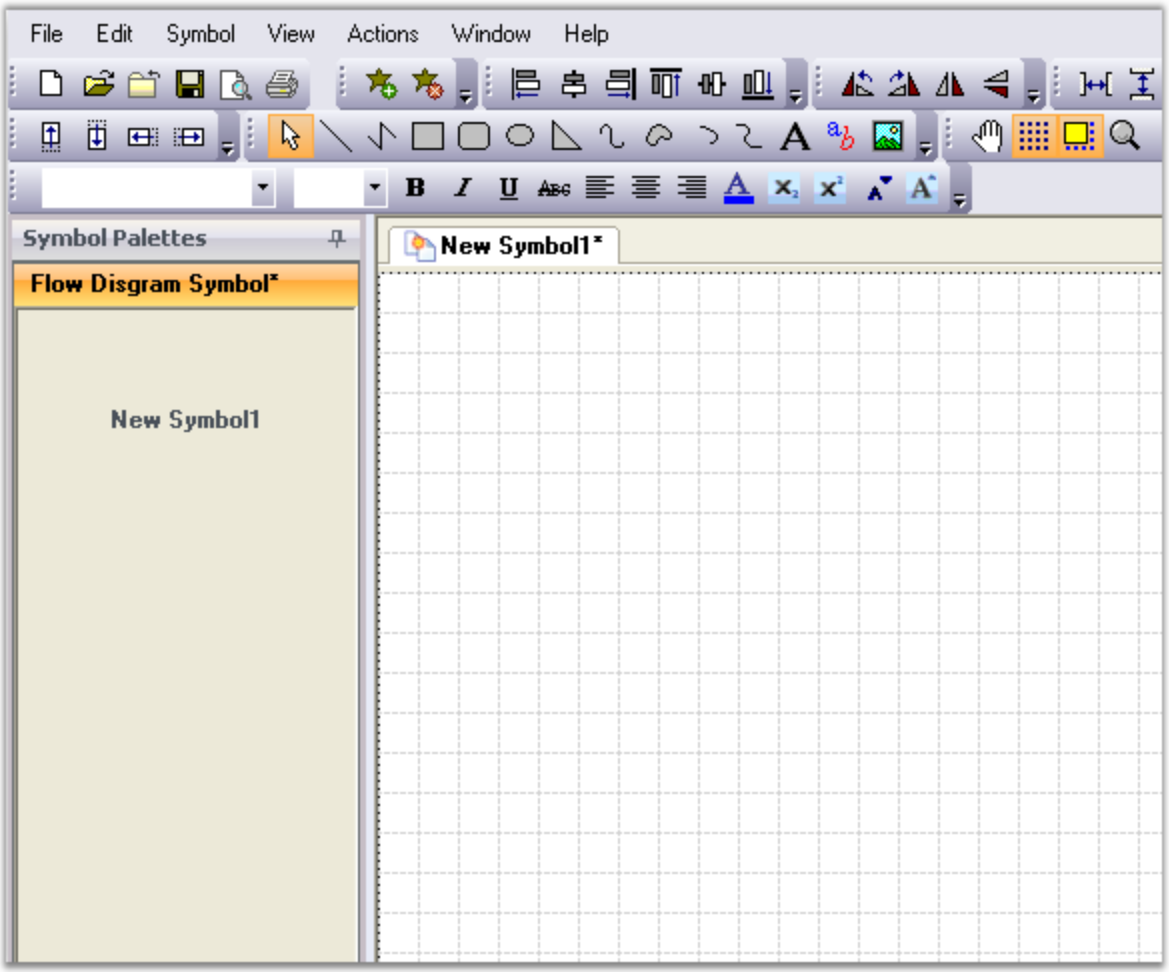

Figure 49: Symbol Designer With New Symbol

- Draw the desired shapes in the grid area. Change the properties of the shapes using **Property** window (Example: Color, Size etc).

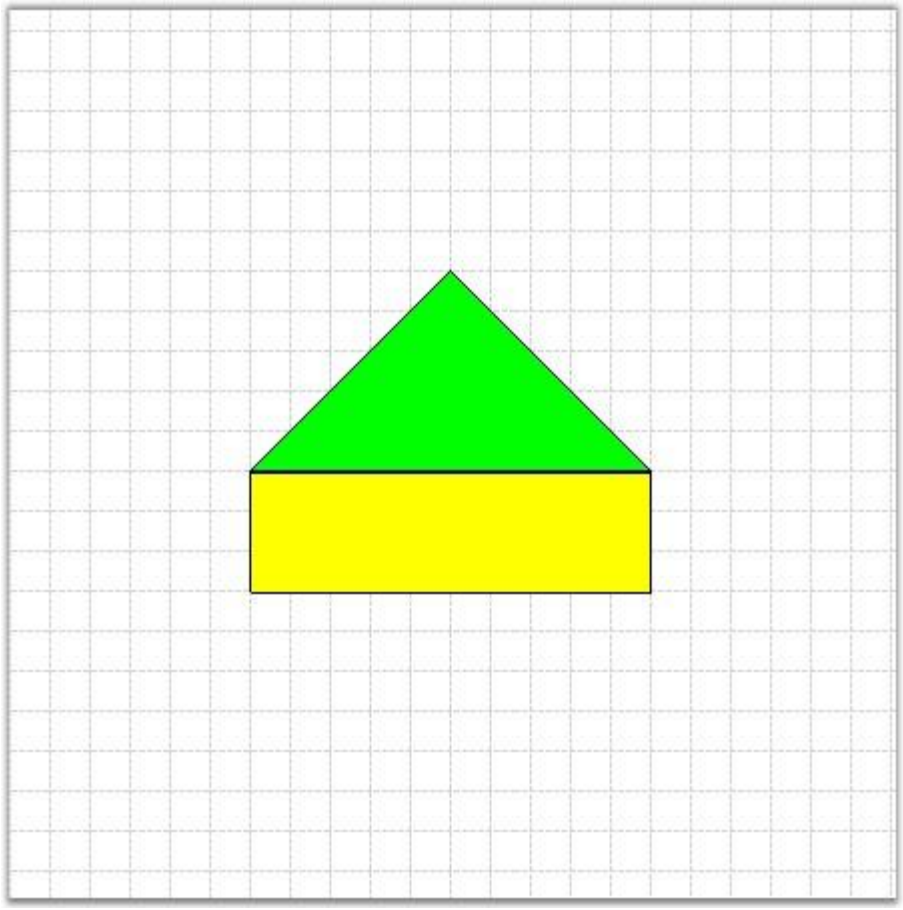

Figure 50: Symbol

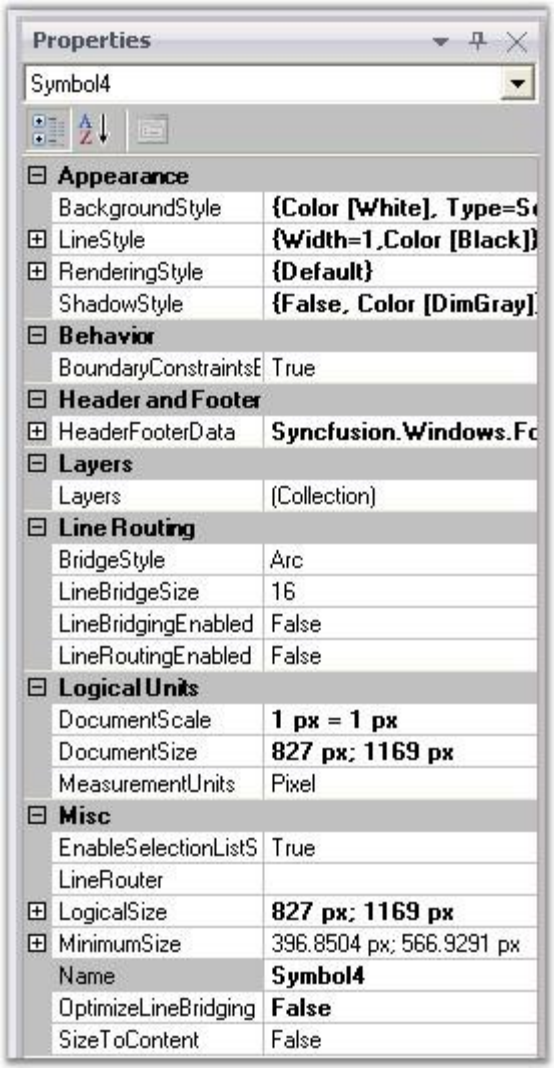

Figure 51: Properties Window

- After modifying the default settings, we have to save this symbol into the symbol palette. Go to the **File** menu and click **Save**. A **Save SumbolPalette** dialog will appear as in the following screen shot.

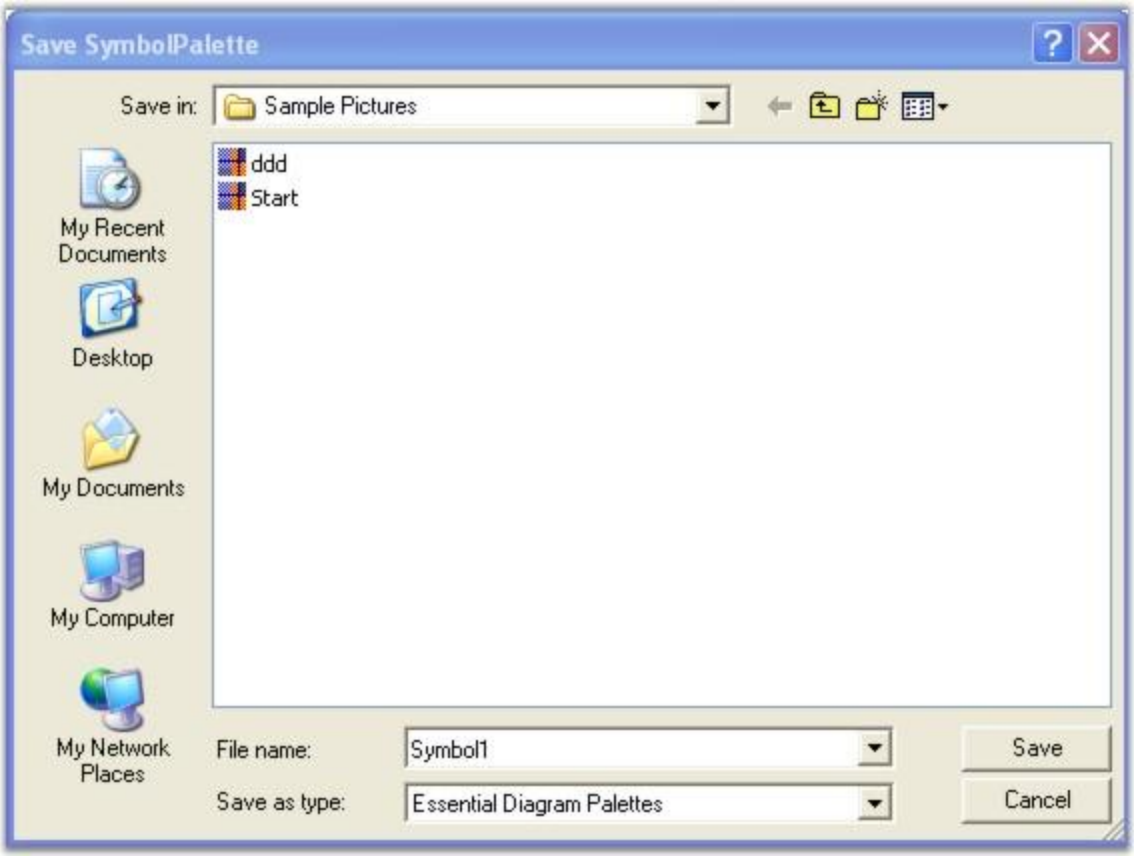

Figure 52: Save SymbolPalette Dialog

- Give a relevant file name for the palette and click **Save**. As the palette is saved, new symbol is automatically loaded into the symbol palette.

- You can modify the custom symbols by using the **Properties** window. Click the New Symbol1 in the Symbol Palettes to view the properties. Change the name of the symbol by using the **Name** property.

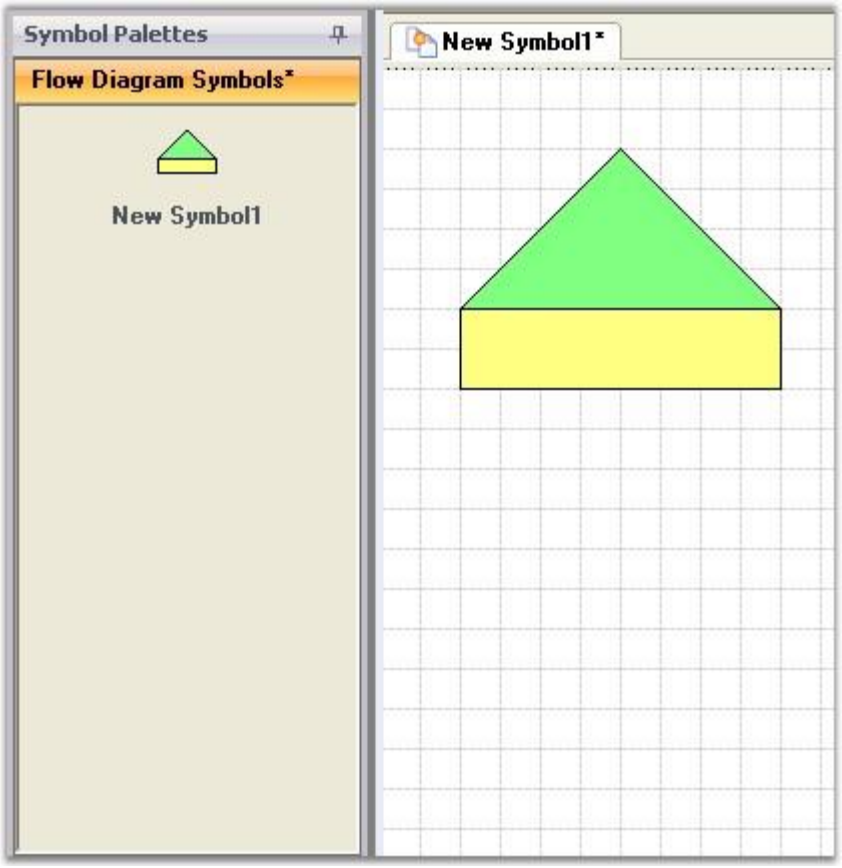

Figure 53: Symbol Palette With New Symbol

- Repeat the steps 3 to 7 for creating more symbols.

- If you create symbols using more than one shapes, you need to group all the shapes into single symbol using the **Group** option in symbol designer.

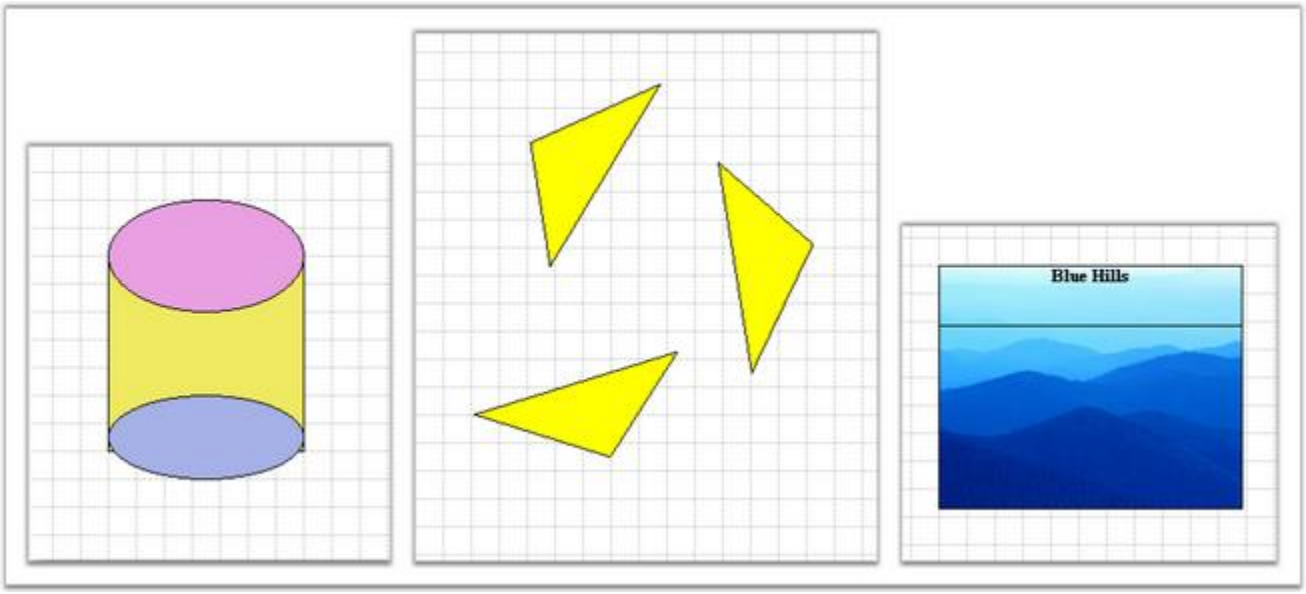

Figure 54: Different Symbols

- The above symbols are added to the symbol palette. You will get a symbol palette as shown in the following screen shot.

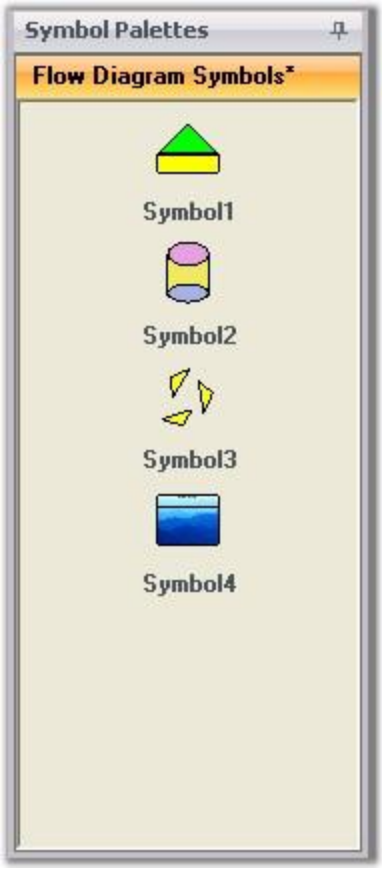

Figure 55: Symbol Palette with New Symbols

- Finally Save the Flow Diagram Symbol Palette. Now the above symbols will be available in .edp file format. You can re-use this in your applications and in the Diagram Builder.

4 Concepts And Features

This section discusses the various concepts and features of the Diagram control in the following sections.

4.1 Supported Controls

The controls supported by Essential Diagram are as follows.

The controls associated with the Diagram control are illustrated in the following image.

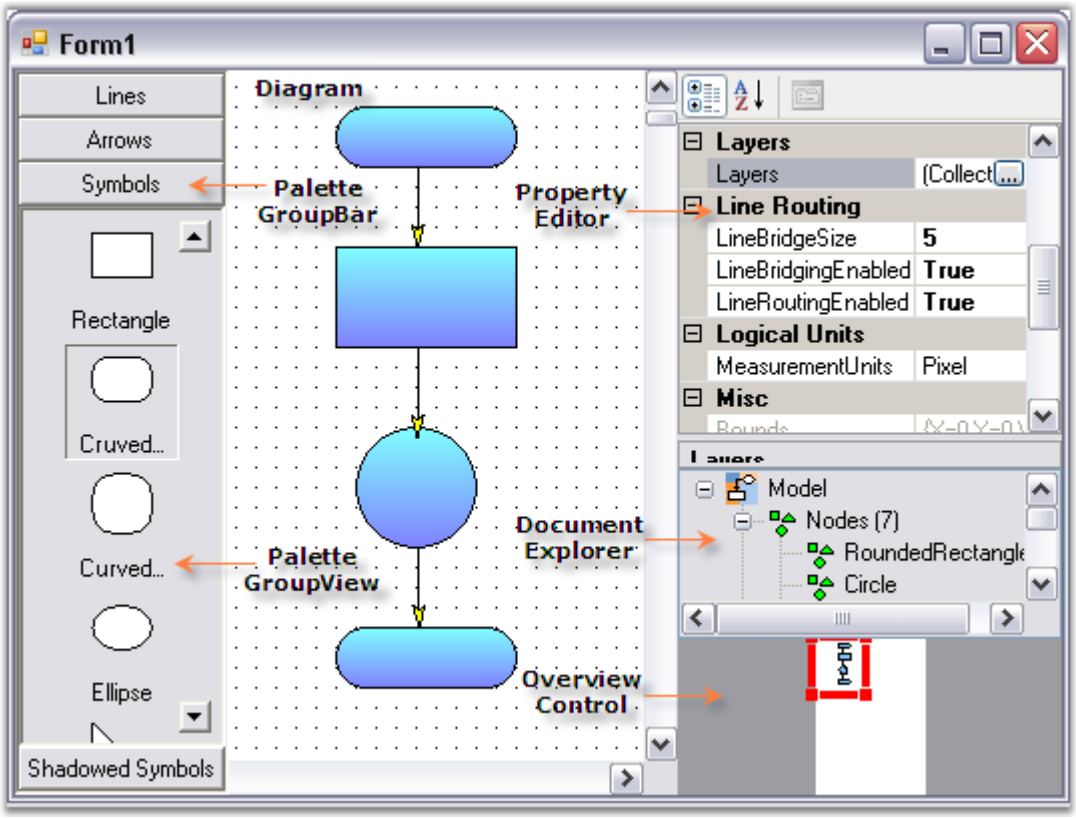

Figure 56: Diagram With Supported Controls

4.1.1 Overview Control

Overview Control provides a perspective view of a diagram model, and allows users to dynamically pan and zoom the diagrams. The control features a view port window that can be moved and / or resized using the mouse to modify the diagrams' origin and magnification properties at run-time.

The important property of the Overview Control is the **Diagram** property. The following are the list of properties of the Overview control.

Property	Description
BackColor	Background color of the component.
AllowDrop	Gets or sets a value indicating whether the control can accept the data that the user can drops on it.
BackgroundImage	Background image of the component.
BorderStyle	Sets the border style for the component. It can be FixedSingle, Fixed3D or None.
Controls	Indicates the collection of control within the component.
Enabled	Indicates if the control is enabled.
Dock	Indicates which control borders are docked to its parent control and determine how the control is resized with its parent.
Diagram	Sets the corresponding diagram to the Overview Control.
Visible	Sets the visibility of the control.

The important events of Overview Control are listed below with their corresponding descriptions.

Event	Description
Click	Occurs when the component is clicked.
DoubleClick	Occurs when the component is double-clicked.
ViewPortBoundsChanged	Occurs when the controls viewport bounds is

	changed.
ViewPortBoundsChanging Event	Occurs when the controls viewport bounds is changing.

Programmatically, the properties can be set as follows.

```
[C#]

overviewControl1.BackColor = System.Drawing.SystemColors.AppWorkspace;
overviewControl1.Diagram = diagram1;
overviewControl1.Dock = System.Windows.Forms.DockStyle.Bottom;
overviewControl1.ForeColor = System.Drawing.Color.Red;
overviewControl1.Location = new System.Drawing.Point(0, 377);
overviewControl1.Name = "overviewControl";
overviewControl1.Size = new System.Drawing.Size(200, 100);
overviewControl1.TabIndex = 1;
```

```
[VB]

overviewControl1.BackColor = System.Drawing.SystemColors.AppWorkspace
overviewControl1.Diagram = diagram1
overviewControl1.Dock = System.Windows.Forms.DockStyle.Bottom
overviewControl1.ForeColor = System.Drawing.Color.Red
overviewControl1.Location = New System.Drawing.Point(0, 377)
overviewControl1.Name = "overviewControl"
overviewControl1.Size = New System.Drawing.Size(200, 100)
overviewControl1.TabIndex = 1
```

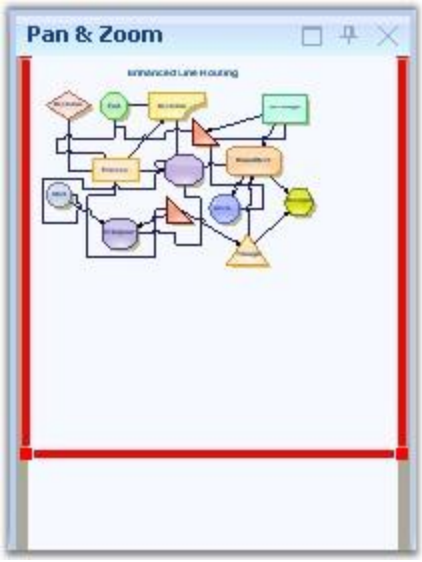

Figure 57: Overview Control

4.1.2 Palette Groupbar And GroupView

The Palette Groupbar control provides a way for users to drag-and-drop the symbols onto a diagram. It is based on the Syncfusion Essential Tools GroupBar control. Each symbol palette loaded in the PaletteGroupBar occupies a panel that can be selected by a bar button. The bar button is labeled with the name of the symbol palette. The symbols in the palette are shown as icons that can be dragged and dropped onto the diagram. This control allows users to add symbols to a palette, and save or load the palette whenever necessary. It provides a way to classify and maintain the symbols.

The PaletteGroupView control provides an easy way to serialize a symbol palette to and from the resource file of a form. At the design-time, users can attach a symbol palette to a PaletteGroupView control in a form. Selecting the PaletteGroupView, and clicking the **Palette** property in the Visual Studio .NET properties window, opens a standard **Open File** dialog, which allows the user to select a symbol palette file that has been created using the Symbol Designer.

The properties of the PaletteGroupBar and GroupView with their descriptions are given in the below table.

Property	Description
BackColor	Sets the background color of the component.

BorderStyle	Sets the border style to FixedSingle, Fixed3D or None.
Collapsed	Indicates whether the GroupBar is collapsed.
CollapsedText	Sets the text for the collapsed client area of the GroupBar.
CollapsedWidth	Specifies the width of the collapsed GroupBar.
CollapseImage	Image of the collapsed button in the expanded state.
DrawClientBorder	Indicates whether border is drawn around the GroupBar's client window.
ExpandImage	Sets image of the collapse button.
FlatLook	Indicates whether control is displayed with a flat look.
Font	Sets font style for text in the control.
ForeColor	Sets fore color of the display text in the component.
GroupBarItemCursor	Cursor that is to be displayed when the mouse pointer is over the GroupBarItems.
Office2007Theme	Sets the (blue, black or silver) office theme used for drawing the control.
PopupClientSize	Sets the initial size of the pop-up for GroupBar client.
PopupResizeMode	Gets / sets the pop-up resize mode.
ShowPopupGripper	Boolean value indicating whether to show GroupBarItem's popup gripper.
Text	Text associated with the control.
TextAlign	Alignment of the text set through Text property.
ThemesEnabled	Specifies whether control should be themed.

VisualStyle	Visual style for drawing the control. Styles are Default, OfficeXP, Office2003, VS2005 and Office2007.
AllowCollapse	Indicates whether GroupBar can be collapsed.
AnimatedSelection	Indicates whether animated selection is enabled.
BarHighlight	Indicates whether GroupBar item is highlighted on mouse hovering over a GroupBar Item.
EditMode	This property determines whether the symbols from the palette can be dragged and dropped onto the Diagram.
Enabled	Indicates whether component is enabled.
ExpandButtonToolTip	Sets tooltip for Collapse button, when the control is collapsed.
GroupBarItemHeight	Height of the GroupBarItems.
MinimizeButtonToolTip	ToolTip for collapse button when control is expanded.
NavigationPaneToolTip	ToolTip for navigation pane.
PopupAutoClose	Indicates whether pop-up is closed after clicking an item.
SelectedItem	Index of the selected GroupBarItem.
StackedMode	Indicates whether GroupBarItem is stacked.
Visible	Sets the visibility of the GroupBar control.
GroupBarItems	GroupBarItem collection in the control.
ShowChevron	Indicates if chevron button of the navigation panel should be displayed when required.
ShowItemImageInHeader	Gets / sets a value indicating whether the selected item's image is shown in the header in Stacked GroupBar.

Palette	Indicates the loaded palette is in palette view.

Method	Description
LoadPalette	Loads given Symbol Palette to the PaletteGroupView.

The important events of the PaletteGroupBar and GroupView with their descriptions are given in the below table.

Event	Description
Click	Occurs when component is clicked.
DoubleClick	Occurs when the component is double-clicked.
GroupViewItemHighlighted	Event fired when an item in the GroupView control is highlighted.
GroupViewItemSelected	Event fired when an item in the GroupView control is selected.
GroupViewItemReordered	Event fired after the GroupView control items have been reordered by a drag–and–drop operation.
GroupViewItemRenamed	Event fired after an in-place rename operation.
ShowContextMenu Event	Event fired when the right mouse button is clicked over the control.

Programmatically, the properties can be set as follows.

```
[C#]

paletteGroupBar1.AllowDrop = true;
paletteGroupBar1.Controls.Add(paletteGroupView1);
paletteGroupBar1.Controls.Add(paletteGroupView2);
paletteGroupBar1.Dock = System.Windows.Forms.DockStyle.Left;
paletteGroupBar1.EditMode = false;
paletteGroupBar1.GroupBarItems.AddRange(new
Syncfusion.Windows.Forms.Tools.GroupBarItem[]
                  { groupBarItem1, groupBarItem2 });
```

```
paletteGroupBar1.Location = new System.Drawing.Point(0, 0);
paletteGroupBar1.Name = "paletteGroupBar1";
paletteGroupBar1.SelectedItem = 1;
paletteGroupBar1.Size = new System.Drawing.Size(114, 477);
paletteGroupBar1.TabIndex = 1;
paletteGroupBar1.Text = "Symbol Palette";

groupBarItem1.Client = paletteGroupView1;
groupBarItem1.Text = "Basic Shapes";
groupBarItem2.Client = paletteGroupView2;
groupBarItem2.Text = "ElectricalSymbols";
paletteGroupView1.ButtonView = true;
paletteGroupView1.Location = new System.Drawing.Point(2, 24);
paletteGroupView1.Name = "paletteGroupView1";
paletteGroupView1.Size = new System.Drawing.Size(71, 0);
paletteGroupView1.TabIndex = 0;

paletteGroupView1.Text = "paletteGroupView1";

paletteGroupView1.LoadPalette
(@"..\..\..\..\..\..\..\..\..\Common\Data\Diagram\BasicShapes.edp");


paletteGroupView2.LoadPalette
(@"..\..\..\..\..\..\..\..\..\Common\Data\Diagram\ElectricalSymbols.edp
");
```

[VB]

```
paletteGroupBar1.AllowDrop = True
paletteGroupBar1.Controls.Add(paletteGroupView1)
paletteGroupBar1.Controls.Add(paletteGroupView2)
paletteGroupBar1.Dock = System.Windows.Forms.DockStyle.Left
paletteGroupBar1.EditMode = False
paletteGroupBar1.GroupBarItems.AddRange(New
Syncfusion.Windows.Forms.Tools.GroupBarItem() {groupBarItem1,
groupBarItem2})


paletteGroupBar1.Location = New System.Drawing.Point(0, 0)
paletteGroupBar1.Name = "paletteGroupBar1"
paletteGroupBar1.SelectedItem = 1
paletteGroupBar1.Size = New System.Drawing.Size(114, 477)
paletteGroupBar1.TabIndex = 1
paletteGroupBar1.Text = "Symbol Palette"
```



```
groupBarItem1.Client = paletteGroupView1
groupBarItem1.Text = "Basic Shapes"
groupBarItem2.Client = paletteGroupView2
groupBarItem2.Text = "ElectricalSymbols"
paletteGroupView1.ButtonView = True
paletteGroupView1.Location = New System.Drawing.Point(2, 24)
paletteGroupView1.Name = "paletteGroupView1"
paletteGroupView1.Size = New System.Drawing.Size(71, 0)
paletteGroupView1.TabIndex = 0

paletteGroupView1.Text = "paletteGroupView1"

paletteGroupView1.LoadPalette("..\..\..\..\..\..\..\..\..\Common\Data\D
iagram\BasicShapes.edp")


paletteGroupView2.LoadPalette("..\..\..\..\..\..\..\..\..\Common\Data\D
iagram\ElectricalSymbols.edp")
```

Dynamically add Symbol Palette into PaletteGroupBar

You can add Symbol Palettes into PaletteGroupBar by means of deserializing the palette (*.edp) file dynamically. The PaletteGroupBar control supports **PaletteGroupBar1.AddPalette()** method in order to add a palette into the PaletteGroupBar.

Follow the steps given below for adding symbol palette into PaletteGroupBar:

1. Add OpenFileDialog control into form.
   4.
2. Set the **Filter** property of OpenFileDialog as,
   5.
3. Essential Diagram Palettes|*.edp|Visio Stencils|*.vss; *.vsx|Visio Drawings(Shapes only)|*.vsd; *.vdx|All files|*.*
   6.
4. Add the following lines of code to your button click event.

```
[C#]

if (openPaletteDialog.ShowDialog(this) == DialogResult.OK)
{
    SymbolPalette curSymbolPalette;
    FileStream iStream;
    string strFileName = openPaletteDialog.FileName;
    RegexOptions options = RegexOptions.IgnoreCase |
```

```
RegexOptions.RightToLeft;
    Match match = Regex.Match(strFileName, ".vss|.vsx|.vsd|.vdx",
options);
    if (match.Success)
    {
        VisioStencilConverter converter = new
VisioStencilConverter(strFileName, this);
        converter.ShowProgressDialog = true;
        curSymbolPalette = converter.Convert();
        if (curSymbolPalette != null)
            PaletteGroupBar1.AddPalette(curSymbolPalette);
    }
    else
    {
        try
        {
            iStream = new FileStream(strFileName, FileMode.Open,
FileAccess.Read);

            // Deserialize  the Binary format
            IFormatter formatter = new BinaryFormatter();
            AppDomain.CurrentDomain.AssemblyResolve +=
                        new
ResolveEventHandler(DiagramBaseAssembly.AssemblyResolver);
            curSymbolPalette =
(SymbolPalette)formatter.Deserialize(iStream);
            PaletteGroupBar1.AddPalette(curSymbolPalette);
        }
        catch (Exception se)
        {
            MessageBox.Show(this, se.Message);
        }
        finally
        {
            iStream.Close();
        }
    }
}
```

Saving the active Palette

You can save the current active palette of PaletteGroupBar window by means of serializing the palette (.edp) file. The **PaletteGroupBar.CurrentSymbolPalette** property returns the currently selected symbol palette.

Follow the steps given below for saving current symbol palette

1. Add SaveFileDialog control into form.
2. Set the **Filter** property of SaveFileDialog as
   7. Essential Diagram Palettes|*.edp|All files|*.*
3. Add the following lines of code to your button click event.

```
[C#]

if (savePaletteDialog.ShowDialog(this) == DialogResult.OK)
{
    SymbolPalette symbolPalette =
PaletteGroupBar1.CurrentSymbolPalette;
    string strSavePath = savePaletteDialog.FileName;

    if (symbolPalette != null)
    {
        FileStream fStream = new FileStream(strSavePath,
FileMode.OpenOrCreate, FileAccess.Write);
        BinaryFormatter formatter = new BinaryFormatter();
        formatter.Serialize(fStream, symbolPalette);
        fStream.Close();
    }
}
```

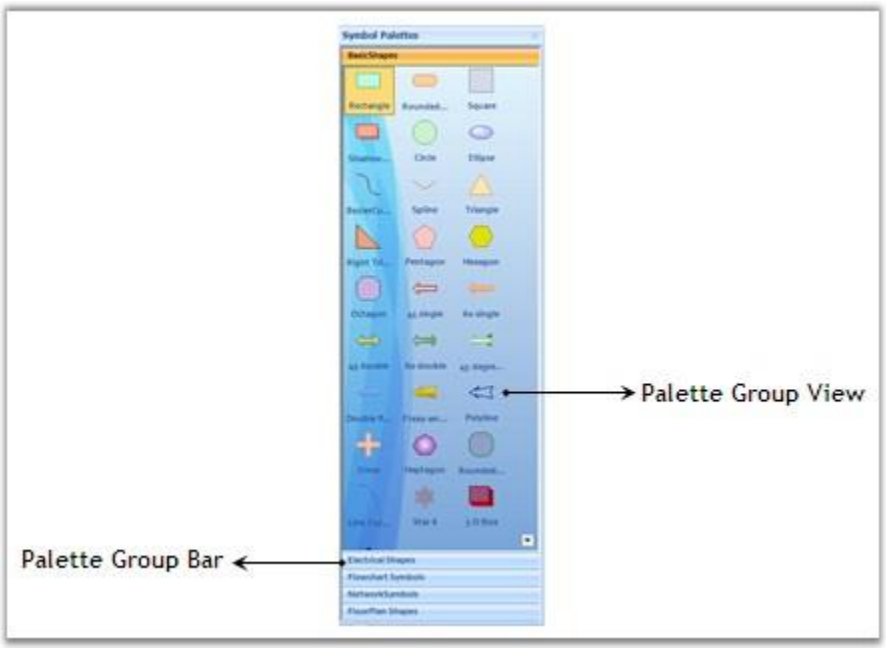

Figure 58: Palette GroupBar and Palette GroupView

4.1.3 Document Explorer

Document Explorer allows you to visualize the details of various objects that are added onto the diagram control at run-time. The layers will be listed under the Layers node and other objects like shapes, links, lines and text editor will be listed under Nodes node.

The properties of the Document Explorer are listed below with their respective descriptions.

Property	Description
BackColor	Background color of the component.
BorderStyle	Border style for the component. It can be FixedSingle, Fixed3D or None.
CheckBoxes	Boolean value indicating whether check boxes should be displayed besides the nodes.
ItemHeight	Height of the tree view node.
Enabled	Indicates if the control is enabled.
FullRowSelect	Indicates whether the whole row (through out the width of the TreeView) is selected when the corresponding node is selected.
HideSelection	Removes the highlight from the selected node when the control loses focus.
HotTracking	Indicates whether the selected node will interact with the user by giving a link-like appearance.
ImageIndex	Default image index for the nodes.
ImageKey	Default image key for the nodes.
Imagelist	Imagelist with images to be used for the nodes.
Indent	Indentation of child nodes in pixels.
LabelEdit	Boolean value indicating whether nodes labels can be edited.

LineColor	Color of the lines that connects the nodes of the TreeView.
Nodes	Node Collection of the TreeView control.
PathSeparator	String Delimiter used for the path returned by a node's Fullpath property.
Scrollable	Enables scroll bars if required.
SelectedImageIndex	Default image index for the selected nodes.
SelectedImageKey	Default image key for the selected nodes.
ShowLines	Indicates whether lines are displayed between sibling nodes and between parent and child nodes.
ShowNodeToolTips	Indicates whether tooltips will be displayed on the nodes.
ShowPlusMinus	Indicates whether plus / minus buttons are shown next to parent nodes.
ShowRootLines	Indicates whether lines are shown between root nodes.
StateImageList	ImageList used for custom state images.
Visible	Sets visibility of the control.

Method	Description
AttachModel	Adds Diagram Model to the Document Explorer.

The important events of Document Explorer are as follows,

Event	Description
Click	Occurs when the component is clicked.
DoubleClick	Occurs when the component is double-clicked.
AfterCheck	Occurs when a check box on a tree node has

	been checked or unchecked.
AfterCollapse	Occurs when a node has been collapsed.
AfterExpand	Occurs when a node has been expanded.
AfterLabelEdit	Occurs when the text of a node has been edited by the user.
AfterSelect	Occurs when the selection has been changed.
BeforeCheck	Occurs when a check box on a tree node is about to be checked or unchecked.
BeforeCollapse	Occurs when a node is about to be collapsed.
BeforeExpand	Occurs when a node is about to be expanded.
BeforeLabelEdit	Occurs when the text of a node is about to be edited by the user.
BeforeSelect	Occurs when the selection is about to change.
DrawNode	Occurs in owner draw-mode, when a node needs to be drawn.
NodeMouseClick	Occurs when a node is clicked with the mouse.
NodeMouseDoubleClick	Occurs when a node is double-clicked with the mouse.

Programmatically, the properties can be set as follows.

```
[C#]

documentExplorer1.AttachModel(model1);
documentExplorer1.Dock = DockStyle.Right;
documentExplorer1.BackColor = System.Drawing.SystemColors.Window;
documentExplorer1.Location = new System.Drawing.Point(0, 377);
documentExplorer1.Size = new System.Drawing.Size(200, 100);
documentExplorer1.BorderStyle =
System.Windows.Forms.BorderStyle.Fixed3D;
documentExplorer1.ShowNodeToolTips = true;
```

```
[VB]

documentExplorer1.AttachModel(model1)
documentExplorer1.Dock = DockStyle.Right
documentExplorer1.BackColor = System.Drawing.SystemColors.Window
documentExplorer1.Location = New System.Drawing.Point(0, 377)
documentExplorer1.Size = New System.Drawing.Size(200, 100)
documentExplorer1.BorderStyle =
System.Windows.Forms.BorderStyle.Fixed3D
documentExplorer1.ShowNodeToolTips = True
```

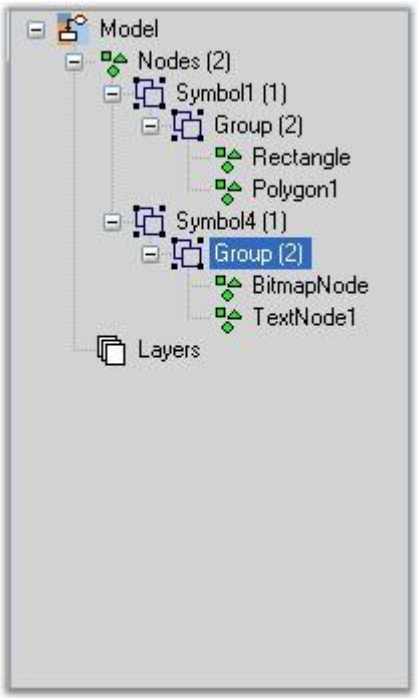

Figure 59: Document Explorer

Sample code snippet for documentExplorer1.AfterSelect Event

```
[C#]

documentExplorer1.AfterSelect+=new TreeViewEventHandler(
documentExplorer1_AfterSelect );

private void documentExplorer1_AfterSelect(object
sender,TreeViewEventArgs e)
{
  // Update diagram's selection list depending on TreeNode Tag
   if ( e.Node.Tag is Node )
```



```
    {
        Node nodeTemp = e.Node.Tag as Node;
        if ( nodeTemp != null )
        {
          if (nodeTemp.Visible &&
nodeTemp.Root.Equals(this.diagram1.Model))
          {
              diagram1.View.SelectionList.Clear();
              diagram1.View.SelectionList.Add(e.Node.Tag as Node);
          }
          else
          {
              propertyEditor.PropertyGrid.SelectedObject = nodeTemp;
          }
        }
     }
}
```

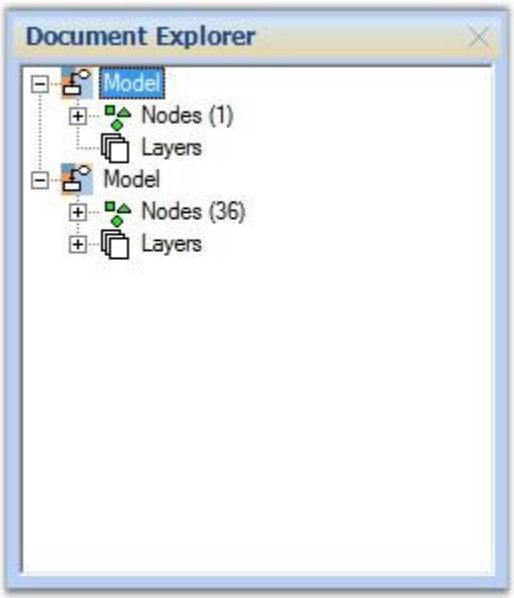

Figure 60: Document Explorer

4.1.4 Property Editor

The Property Editor in Essential Diagram displays properties of the currently selected object(s) in the diagram. It is a windows forms control that can be added to the Visual Studio .NET toolbox. It also allows the users to set or modify various properties of the objects or the model. The Property Editor provides an easy interface, to set and view various property settings.

The following table lists the properties of the Property Editor. The important property of the Property Editor is the **Diagram** property.

Property	Description
Diagram	It contains a reference to the diagram that this property editor is attached to. The property editor receives events from the diagram when the current selection changes. It updates the currently displayed object in the property editor.
Product Name	Gets the product name of the assembly containing the control.
ProductVersion	Gets the version of the assembly containing the control.
PropertyGrid	Gets the reference to the PropertyGrid object contained by this property editor.
ShowCombo	Determines if combo box is visible.

Programmatically, the properties can be set as follows.

```
[C#]

this.propertyEditor.PropertyGrid.BackColor =
System.Drawing.Color.FromArgb((( System.Byte)(227)),
((System.Byte)(239)),((System.Byte)(255)));
this.propertyEditor.PropertyGrid.CommandsBackColor =
System.Drawing.Color.FromArgb(((System.Byte)(227)),((System.Byte)(239))
,((System.Byte)(255)));
this.propertyEditor.PropertyGrid.CommandsForeColor=
System.Drawing.Color.MidnightBlue;
this.propertyEditor.PropertyGrid.Font = new
System.Drawing.Font("Arial",
8.25F,System.Drawing.FontStyle.Regular,System.Drawing.GraphicsUnit.Poin
t,(System.Byte)(0)));
this.propertyEditor.PropertyGrid.HelpBackColor =
System.Drawing.Color.FromArgb(((System.Byte)(227)),((System.Byte)239)),
((System.Byte)(255)));
```



```
this.propertyEditor.PropertyGrid.HelpForeColor =
System.Drawing.Color.MidnightBlue;
this.propertyEditor.PropertyGrid.LineColor =
System.Drawing.Color.FromArgb((( System.Byte)(185)),
((System.Byte)(216)), ((System.Byte)(255)));
this.propertyEditor.PropertyGrid.ViewBackColor =
System.Drawing.Color.FromArgb(((System.Byte)(227)), ((
System.Byte)(239)), ((System.Byte)(255)));
this.propertyEditor.PropertyGrid.ViewForeColor=
System.Drawing.Color.MidnightBlue;
this.propertyEditor.ShowCombo = true;
this.propertyEditor.Diagram = diagram1;
```

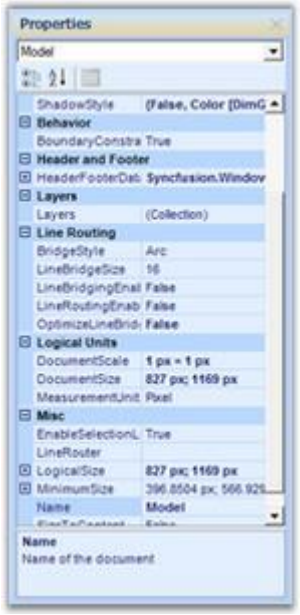

Figure 61: Property Editor

4.1.5 Diagram Grid

Diagram Grid is drawn with evenly spaced points that provides a visual guidance to the user.

Behavior

Draws a matrix of evenly spaced points in the view, and provides snap to the grid calculations.

Class Reference

It is a property of **Diagram.View** class and its return type is ***Syncfusion.Windows.Forms.Diagram.LayoutGrid*.**

Properties

Properties	Description
Color	Color used for drawing the grid. It accepts System.Color value.
ContainerView	Gets or sets the view that this grid is attached to.
DashOffset	Distance from the start of the line to the dash pattern. It accepts Float value.
DashStyle	Style used for dashed lines. It accepts System.Drawing.Drawing2D.DashStyle value.
GridStyle	Gets or sets the appearance of the grid. It is GridStyle enumerator type value.
HorizontalSpacing	Determines the horizontal distance between grid points. It accepts float value.
MinPixelSpacing	Indicates minimum spacing between grid points in device units. It accepts Float value.
SnapToGrid	Adjust the node with nearest grid point. Specifies whether the snap to grid feature is enabled. It accepts Boolean value (true or false).
VerticalSpacing	Determines the vertical distance between grid points. It accepts Float value.
Visible	Specifies whether the grid is visible. It accepts Boolean value (true or false).

[C#]

```
diagram1.View.Grid.GridStyle = GridStyle.Line;
diagram1.View.Grid.DashStyle=System.Drawing.Drawing2D.DashStyle.Dot;
diagram1.View.Grid.Color = Color.LightGray
diagram1.View.Grid.VerticalSpacing = 15;
diagram1.View.Grid.HorizontalSpacing = 15;
```

```
diagram1.View.Grid.Visible = false;
diagram1.View.Grid.SnapToGrid = true;
```

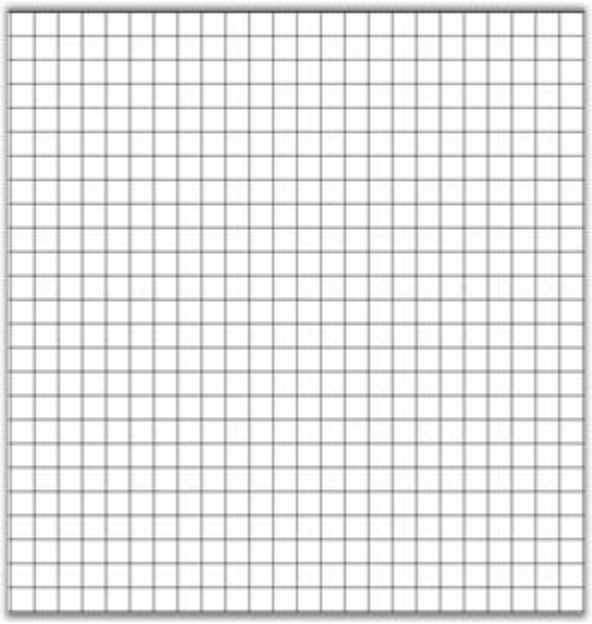

Figure 62: Diagram Grid

4.2 Nodes or Shapes

The Diagram control supports different kinds of nodes. The following are the nodes that are supported by the Diagram control:

- TextNode
- Shape
- Symbol
- ControlNode
- PathNode
- BitmapNode
- RichTextNode
- MetafileNode
- Group
- PseudoGroup
- FilledPath
- FilledShape
- RoundRect
- PolygonA
- Rectangle
- ClosedCurveNode

- Ellipse
- SemiCircle
- CircularArc
- PolylineNode
- CurveNode
- Line
- SplineNode
- Arc
- BezierCurve
- MeasureLine
- Polyline
- OrthogonalConnector
- LineConnector
- OrthogonalLine
- Link
- PolyLineConnector

Creating a Node in the Diagram Control at Run Time

To create a node in the Diagram control:

1. Drag the Diagram control to the windows form.
2. Press the **F7** key to open the ***.cs** file and enter the following code in the **Page_Load** function.

```
[C#]

private void Form1_Load(object sender, EventArgs e)
{
    Syncfusion.Windows.Forms.Diagram.Ellipse ellipse = new
Syncfusion.Windows.Forms.Diagram.Ellipse(10, 10, 110, 70);
    diagram1.Model.AppendChild(ellipse);
}
```

```
[VB]

Private Sub Form1_Load(ByVal sender As Object, ByVal e As EventArgs)
    Dim ellipse As New Syncfusion.Windows.Forms.Diagram.Ellipse(10, 10,
110, 70)
    diagram1.Model.AppendChild(ellipse)
End Sub
```


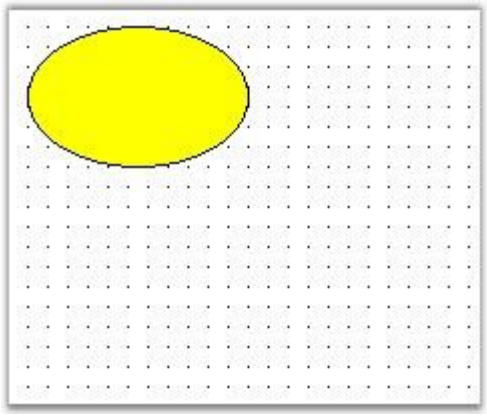

Figure 63: Diagram with Node Sample

Node Property Settings

The following code example demonstrates how to apply node property settings.

```
[C#]

private void Form1_Load(object sender, EventArgs e)
{
    Syncfusion.Windows.Forms.Diagram.Ellipse ellipse = new
Syncfusion.Windows.Forms.Diagram.Ellipse(10, 10, 110, 70);
    diagram1.Model.AppendChild(ellipse);
    ellipse.FillStyle.Color = System.Drawing.Color.AliceBlue;
    ellipse.FillStyle.ColorAlphaFactor = 100;
    ellipse.FillStyle.ForeColor = System.Drawing.Color.Aquamarine;
    ellipse.FillStyle.ForeColorAlphaFactor = 70;
    ellipse.FillStyle.Type = FillStyleType.PathGradient;
    ellipse.FillStyle.PathBrushStyle =
PathGradientBrushStyle.RectangleCenter;
    ellipse.FillStyle.Type = FillStyleType.LinearGradient;
    ellipse.FillStyle.GradientAngle = 95;
    ellipse.FillStyle.GradientCenter = 0.5f;

    ellipse.EditStyle.AllowChangeHeight = true;
    ellipse.EditStyle.AllowChangeWidth = true;
    ellipse.EditStyle.AllowDelete = false;
    ellipse.EditStyle.AllowMoveX = true;
    ellipse.EditStyle.AllowMoveY = false;
    ellipse.EditStyle.AllowRotate = false;
    ellipse.EditStyle.AllowSelect = true;
}
```

```
[VB]

Private Sub Form1_Load(ByVal sender As Object, ByVal e As EventArgs)
    Dim ellipse As New Syncfusion.Windows.Forms.Diagram.Ellipse(10, 10,
110, 70)
    diagram1.Model.AppendChild(ellipse)
    ellipse.FillStyle.Color = System.Drawing.Color.AliceBlue
    ellipse.FillStyle.ColorAlphaFactor = 100
    ellipse.FillStyle.ForeColor = System.Drawing.Color.Aquamarine
    ellipse.FillStyle.ForeColorAlphaFactor = 70
    ellipse.FillStyle.Type = FillStyleType.PathGradient
    ellipse.FillStyle.PathBrushStyle =
PathGradientBrushStyle.RectangleCenter
    ellipse.FillStyle.Type = FillStyleType.LinearGradient
    ellipse.FillStyle.GradientAngle = 95
    ellipse.FillStyle.GradientCenter = 0.5F

    ellipse.EditStyle.AllowChangeHeight = True
    ellipse.EditStyle.AllowChangeWidth = True
    ellipse.EditStyle.AllowDelete = False
    ellipse.EditStyle.AllowMoveX = True
    ellipse.EditStyle.AllowMoveY = False
    ellipse.EditStyle.AllowRotate = False
    ellipse.EditStyle.AllowSelect = True
End Sub
```

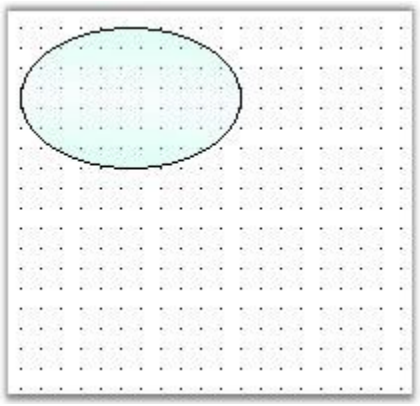

Figure 64: Diagram With Node Property Settings

Creating Nodes and Links

The following code example illustrates how to create nodes and links.

```
[C#]
```

```
protected void Page_Load(object sender, EventArgs e)
{
    Syncfusion.Windows.Forms.Diagram.Ellipse ellipse = new
Syncfusion.Windows.Forms.Diagram.Ellipse(10, 10, 110, 70);
    Syncfusion.Windows.Forms.Diagram.Rectangle rectangle = new
Syncfusion.Windows.Forms.Diagram.Rectangle(250, 50, 50, 80);
    Syncfusion.Windows.Forms.Diagram.LineConnector lineconnector = new
Syncfusion.Windows.Forms.Diagram.LineConnector(new
System.Drawing.PointF(10, 200), new System.Drawing.PointF(300, 250));
    this.diagram1.Model.AppendChild(ellipse);
    this.diagram1.Model.AppendChild(rectangle);
    this.diagram1.Model.AppendChild(lineconnector);
}
```

[VB]

```
Protected Sub Page_Load(ByVal sender As Object, ByVal e As EventArgs)
    Dim ellipse As New Syncfusion.Windows.Forms.Diagram.Ellipse(10, 10,
110, 70)
    Dim rectangle As New
Syncfusion.Windows.Forms.Diagram.Rectangle(250, 50, 50, 80)
    Dim lineconnector As New
Syncfusion.Windows.Forms.Diagram.LineConnector(New
System.Drawing.PointF(10, 200), New System.Drawing.PointF(300, 250))
    Me.diagram1.Model.AppendChild(ellipse)
    Me.diagram1.Model.AppendChild(rectangle)
    Me.diagram1.Model.AppendChild(lineconnector)
End Sub
```

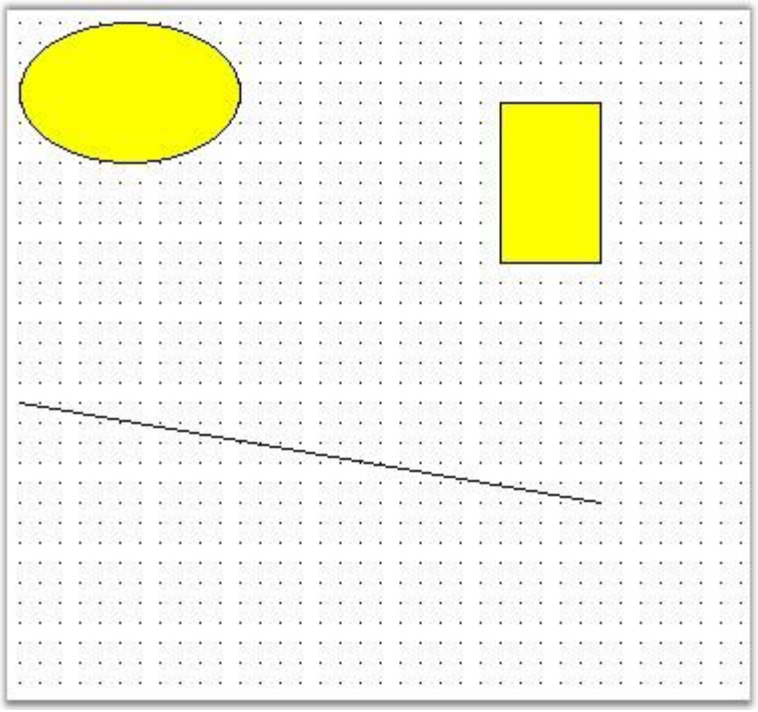

Figure 65: Diagram with Nodes and Links

4.3 Connectors or Links

Connectors and lines have the following decorators:

- Circle
- CircleCross
- CirclereverseArrow
- Cross45
- Cross90
- CrossreverseArrow
- Custom
- Diamond
- DimensionLine
- DoubleArrow
- DoubleCross
- Filled45Arrow
- Filled60Arrow
- FilledCircle
- FilledDiamond
- FilledFancyArrow
- FilledSquare
- None

- Open45Arrow
- Open60Arrow
- OpenFancyArrow
- ReverseArrow
- ReverseDoubleArrow
- Square

Connecting Two Nodes with Line Connector

The following code example illustrates how to create links between two nodes.

```
[C#]

protected void Page_Load(object sender, EventArgs e)
{
    Syncfusion.Windows.Forms.Diagram.Ellipse ellipse = new
Syncfusion.Windows.Forms.Diagram.Ellipse(10, 10, 110, 70);
    Syncfusion.Windows.Forms.Diagram.Rectangle rectangle = new
Syncfusion.Windows.Forms.Diagram.Rectangle(300, 50, 50, 80);
    Syncfusion.Windows.Forms.Diagram.LineConnector lineconnector = new
Syncfusion.Windows.Forms.Diagram.LineConnector(new
System.Drawing.PointF(10, 200), new System.Drawing.PointF(300, 250));
    this.DiagramWebControl1.Model.AppendChild(ellipse);
    this.DiagramWebControl1.Model.AppendChild(rectangle);
    ellipse.CentralPort.TryConnect(lineconnector.HeadEndPoint);
    rectangle.CentralPort.TryConnect(lineconnector.TailEndPoint);
    this.DiagramWebControl1.Model.AppendChild(lineconnector);
}
```

```
[VB]

Protected Sub Page_Load(ByVal sender As Object, ByVal e As EventArgs)
    Dim ellipse As New Syncfusion.Windows.Forms.Diagram.Ellipse(10, 10,
110, 70)
    Dim rectangle As New
Syncfusion.Windows.Forms.Diagram.Rectangle(300, 50, 50, 80)
    Dim lineconnector As New
Syncfusion.Windows.Forms.Diagram.LineConnector(New
System.Drawing.PointF(10, 200), New System.Drawing.PointF(300, 250))
    Me.DiagramWebControl1.Model.AppendChild(ellipse)
    Me.DiagramWebControl1.Model.AppendChild(rectangle)
    ellipse.CentralPort.TryConnect(lineconnector.HeadEndPoint)
    rectangle.CentralPort.TryConnect(lineconnector.TailEndPoint)

    Me.DiagramWebControl1.Model.AppendChild(lineconnector)
End Sub
```

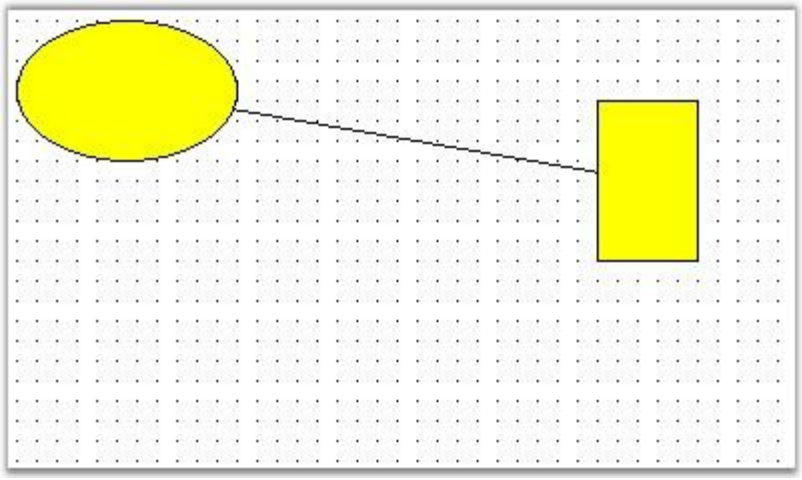

Figure 66: Connection Property Settings

You can change the appearance of the connectors using its properties through code. The following code example illustrates the line properties.

[C#]

```
protected void Page_Load(object sender, EventArgs e)
{

    Syncfusion.Windows.Forms.Diagram.Ellipse ellipse = new
Syncfusion.Windows.Forms.Diagram.Ellipse(160, 60, 100, 60);
    Syncfusion.Windows.Forms.Diagram.Rectangle rectangle = new
Syncfusion.Windows.Forms.Diagram.Rectangle(150, 250, 120, 100);
    Syncfusion.Windows.Forms.Diagram.LineConnector lineconnector = new
Syncfusion.Windows.Forms.Diagram.LineConnector(new
System.Drawing.PointF(10, 200), new System.Drawing.PointF(300, 250));
    this.diagram1.Model.AppendChild(ellipse);
    this.diagram1.Model.AppendChild(rectangle);
    ellipse.CentralPort.TryConnect(lineconnector.TailEndPoint);
    rectangle.CentralPort.TryConnect(lineconnector.HeadEndPoint);
    this.diagram1.Model.AppendChild(lineconnector);
    lineconnector.HeadDecorator.DecoratorShape =
DecoratorShape.Filled45Arrow;
    lineconnector.LineStyle.LineColor = Color.MidnightBlue;
    lineconnector.HeadDecorator.FillStyle.Color = Color.MidnightBlue;
    lineconnector.HeadDecorator.Size = new SizeF(10, 5);
}
```

[VB]


```
Protected Sub Page_Load(ByVal sender As Object, ByVal e As EventArgs)

    Dim ellipse As New Syncfusion.Windows.Forms.Diagram.Ellipse(160,
60, 100, 60)
    Dim rectangle As New
Syncfusion.Windows.Forms.Diagram.Rectangle(150, 250, 120, 100)
    Dim lineconnector As New
Syncfusion.Windows.Forms.Diagram.LineConnector(New
System.Drawing.PointF(10, 200), New System.Drawing.PointF(300, 250))
    Me.diagram1.Model.AppendChild(ellipse)
    Me.diagram1.Model.AppendChild(rectangle)
    ellipse.CentralPort.TryConnect(lineconnector.TailEndPoint)
    rectangle.CentralPort.TryConnect(lineconnector.HeadEndPoint)
    Me.diagram1.Model.AppendChild(lineconnector)
    lineconnector.HeadDecorator.DecoratorShape =
DecoratorShape.Filled45Arrow
    lineconnector.LineStyle.LineColor = Color.MidnightBlue
    lineconnector.HeadDecorator.FillStyle.Color = Color.MidnightBlue
    lineconnector.HeadDecorator.Size = New SizeF(10, 5)
End Sub
```

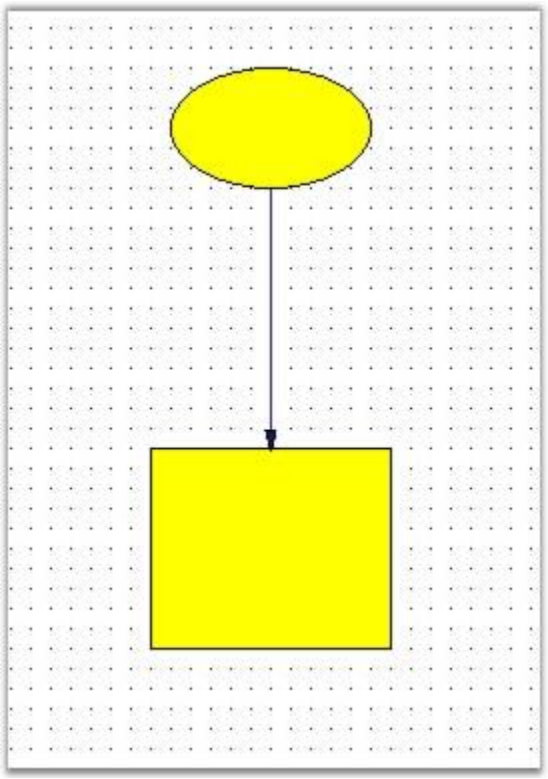

Figure 67: Diagram with Connector Property Settings

4.3.1 Rounded Corner

You can now change the look of connectors (Orthogonal, Org line, Polyline) by providing rounded corners to connectors.

The **EnableRoundedCorner** is used to to enable rounded corner for a connector, and the **CurveRadius** connector property is used and set the radius for the rounded corner curve respectively.

Use Case Scenario

This is used to change the visual style of connectors.

Properties

Property	Description	Data Type
EnableRoundedCorner	Enables or disables rounded corner for a connector.	bool
CurveRadius	Gets or sets the radius for the rounded corner curve of a connector.	float

The following code illustrates how to change the look of a connector by using the EnableRoundedCorner and CurveRadius properties.

```
[C#]
OrthogonalConnector orthogonal = new OrthogonalConnector(new PointF(100, 100), new PointF(300, 300));
// Enables rounded corner for the connector.
orthogonal.EnableRoundedCorner = true;
// Sets the radius of the rounded corner curve.
orthogonal.CurveRadius = 10;
diagram1.Model.AppendChild(orthogonal);
```

```
[VB]
Dim orthogonal As New OrthogonalConnector(New PointF(100, 100), New PointF(300, 300))
'Enables rounded corner for the connector.
orthogonal.EnableRoundedCorner = True
'Sets the radius of the rounded corner curve.
orthogonal.CurveRadius = 10
diagram1.Model.AppendChild(orthogonal)
```

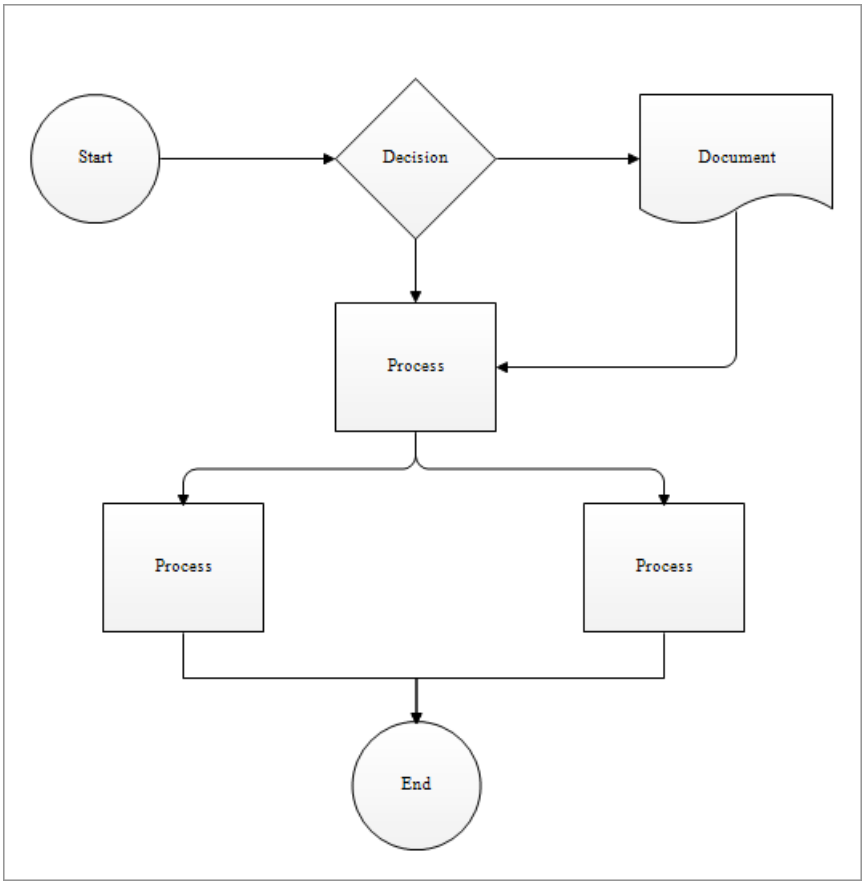

Figure 68: Connectors with Rounded Corners

4.3.2 Drawing Tool

Diagram control now provides a drawing tool to draw the Org line connector dynamically during run time.

Use Case Scenario

The OrgLineConnectorTool is used to draw the Org line connector dynamically.

The following code illustrates how to activate the Org line connector tool:

```
[C#]
// Activates the Org line connector tool.
diagram1.Controller.ActivateTool("OrgLineConnectorTool");
```

```
[VB]
'Activates the Org line connector tool.
diagram1.Controller.ActivateTool("OrgLineConnectorTool")
```

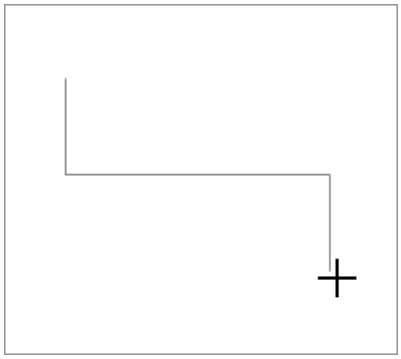

Figure 69: Org line Connector

4.4 Customizing Nodes

This section discusses how to customize Diagram nodes, under the following topics:

4.4.1 Line Bridging

Line bridging provides the visual effect such that the links jump over other links that are found in it's way with lower ZOrder, thereby avoiding the links from intersecting each other and providing a hassle-free view to clearly state the various connections between the nodes. This is done by enabling the LineBridgingEnabled property. Default value is ***false***.

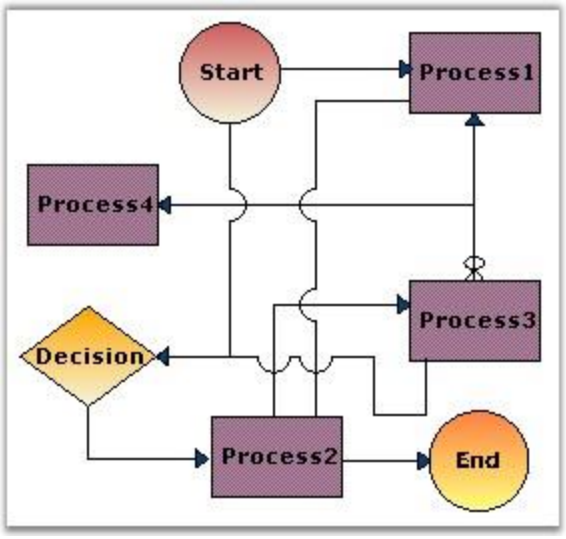

Figure 70: Line Bridging

The below table lists the properties which controls the appearance of the bridge.

Property	Description
LineBridgeSize	Allows to set the size of the bridge when the links intersect each other. Default value is 16.
BridgeStyle	Specifies the type of bridge to be applied. Default value is 'Arc'. The value when set, applies to all the links that are drawn on the diagram. The links will bridge over the other link only when it's ZOrder value is high. The options include the following: • Arc • Gap • Square • Side2 • Side3 • Side4 • Side5 • Side6 • Side7

Programmatically it can be set as follows:

```
[C#]
```



```
this.diagram1.Model.LineBridgeSize = 5;

//enabling for model
this.diagram1.Model.LineBridgingEnabled = true;

//enabling for link object
link.LineBridgingEnabled = true;

this.diagram1.Model.BridgeStyle = BridgeStyle.Square;
```

```
[VB]

Me.diagram1.Model.LineBridgeSize = 5

'enabling for model
Me.diagram1.Model.LineBridgingEnabled = True

'enabling for link object
link.LineBridgingEnabled = True

Me.diagram1.Model.BridgeStyle = BridgeStyle.Square
```

Note: In the above code snippets, link refers to the instance of the Link node.

4.4.2 Line Routing

When a link is drawn between two nodes, by enabling the **LineRoutingEnabled** property of that link and the diagram view, and if any other node is found in between them, the line will be automatically re-routed around those nodes.

Property	Description
LineRoutingEnabled	Specifies whether the links must be re-routed when nodes are found in the path. Default value is false.

Programmatically it can be set as follows:

```
[C#]

//enabling for model
```

```
this.diagram1.Model.LineRoutingEnabled = true;

//enabling for link object
link.LineRoutingEnabled = true;
```

```
[VB]

'enabling for model
Me.diagram1.Model.LineRoutingEnabled = True

'enabling for link object
link.LineRoutingEnabled = True.Model.LineBridgeSize = 5
```

Note:

In the above code snippet, link refers to the instance of the Link node.

Only when LineRoutingEnabled property is set to true, LineRouter properties will be enabled.

8.

Distance and Routing mode settings

To customize the distance between the connectors and the obstacles, and the type of routing to use, the **LineRouter** collection property should be handled. The below properties are available for the LineRouter Collection property.

Line Router Property	Description
DistanceToObstacle	Specifies the distance from routing connector to the obstacle.
RoutingMode	Specifies the type of LineRouting engine routing mode to be used. The default value is 'Inactive'. The options includes, Inactive, Automatic and SemiAutomatic.

Programmatically it can be set as follows.

```
[C#]
```



```
this.diagram1.Model.LineRouter.DistanceToObstacles = 20;
this.diagram1.Model.LineRouter.RoutingMode = RoutingMode.Automatic;
```

```
[VB]

Me.diagram1.Model.LineRouter.DistanceToObstacles = 20
Me.diagram1.Model.LineRouter.RoutingMode = RoutingMode.Automatic
```

The **LineBridgingEnabled**, **LineRoutingEnabled** properties can be set for the diagram, in which case it will be automatically applied to all the links added to the model. Else it can be enabled only for the required links individually.

Node settings

When line routing is enabled make sure to set the **TreatAsObstacle** property of the objects to true, to avoid the links running over them. If not set for an object, then that node will not be considered as an obstacle and the link will pass over it.

Programmatically it can be set as follows:

```
[C#]

circle.TreatAsObstacle = true;
```

```
[VB]

circle.TreatAsObstacle = True
```

In the above code snippets, the TreatAsObstacle property is set to the circle object.

4.4.3 Grouping

A group is a node that acts as a transparent container for other nodes. A group is a composite node that controls a set of child nodes. The bounding rectangle of a group is the union of the bounds of its children. The group renders itself by iterating through its children and rendering them. Child nodes cannot be selected or manipulated individually. Members of the group are added and removed through the **ICompositeNode** interface.

The following code snippet creates a group with two nodes.

```
[C#]
```



```
//Node 1
Syncfusion.Windows.Forms.Diagram.Rectangle nodeRect = new
Syncfusion.Windows.Forms.Diagram.Rectangle(50, 100, 125, 75);
nodeRect.FillStyle.Color = Color.FromArgb(255, 223, 189);
nodeRect.LineStyle.LineColor = Color.Orange;
Syncfusion.Windows.Forms.Diagram.Label lbl = new
Syncfusion.Windows.Forms.Diagram.Label(nodeRect, "Rectangle");
lbl.FontStyle.Size = 12;
lbl.FontStyle.Bold = true;
nodeRect.Labels.Add(lbl);

//Node 2
Syncfusion.Windows.Forms.Diagram.Rectangle nodeRect1 = new
Syncfusion.Windows.Forms.Diagram.Rectangle(150, 100, 125, 75);
nodeRect1.FillStyle.Color = Color.FromArgb(255, 223, 189);
nodeRect1.LineStyle.LineColor = Color.Orange;
Syncfusion.Windows.Forms.Diagram.Label lbl1 = new
Syncfusion.Windows.Forms.Diagram.Label(nodeRect1, "Rectangle1");
lbl1.FontStyle.Size = 12;
lbl1.FontStyle.Bold = true;
nodeRect1.Labels.Add(lbl1);

//Grouping Nodes
Syncfusion.Windows.Forms.Diagram.Group grp = new Group();
grp.AppendChild(nodeRect);
grp.AppendChild(nodeRect1);
this.diagramControl1.Model.AppendChild(grp);
```

[VB.NET]

```
'Node 1
Dim nodeRect As Syncfusion.Windows.Forms.Diagram.Rectangle = New
Syncfusion.Windows.Forms.Diagram.Rectangle(50, 100, 125, 75)
nodeRect.FillStyle.Color = Color.FromArgb(255, 223, 189)
nodeRect.LineStyle.LineColor = Color.Orange
Dim lbl As Syncfusion.Windows.Forms.Diagram.Label = New
Syncfusion.Windows.Forms.Diagram.Label(nodeRect, "Rectangle")
lbl.FontStyle.Size = 12
lbl.FontStyle.Bold = True
nodeRect.Labels.Add(lbl)

'Node 2
Dim nodeRect1 As Syncfusion.Windows.Forms.Diagram.Rectangle = New
Syncfusion.Windows.Forms.Diagram.Rectangle(150, 100, 125, 75)
nodeRect1.FillStyle.Color = Color.FromArgb(255, 223, 189)
nodeRect1.LineStyle.LineColor = Color.Orange
```



```
Dim lbl1 As Syncfusion.Windows.Forms.Diagram.Label = New
Syncfusion.Windows.Forms.Diagram.Label(nodeRect1, "Rectangle1")
lbl1.FontStyle.Size = 12
lbl1.FontStyle.Bold = True
nodeRect1.Labels.Add(lbl1)

'Grouping Nodes
Dim grp As Syncfusion.Windows.Forms.Diagram.Group = New
Syncfusion.Windows.Forms.Diagram.Group()
grp.AppendChild(nodeRect)
grp.AppendChild(nodeRect1)
Me.diagramWebControl1.Model.AppendChild(grp)
```

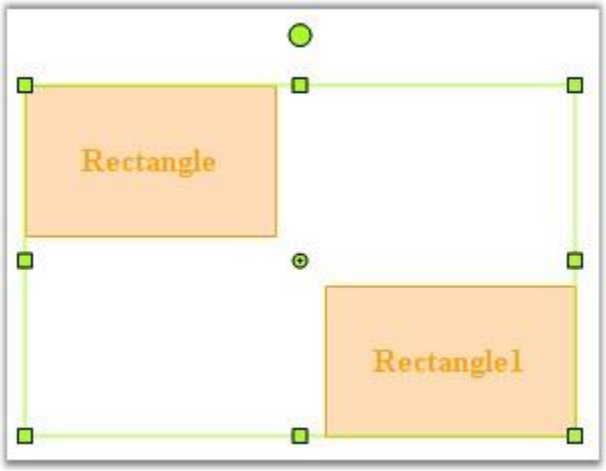

Figure 71: Selecting a Group in a Diagram

The grouping of nodes can be performed using ready-made API.

```
[C#]

//Method to Group the nodes
this.diagram1.Controller.Group();

'Method to UnGroup the nodes
this.diagram1.Controller.UnGroup();
```

```
[VB.NET]

//Method to Group the nodes
Me.diagram1.Controller.Group()
```



```
'Method to UnGroup the nodes
Me.diagram1.Controller.UnGroup()
```

4.4.4 Label

A label is a text object that is attached to a node and is positioned relative to node coordinates. This enables you to format the label text. You can also customize the appearance of the label. You can bind the node name to the label so that the name will be displayed as the label text. This can be achieved using the *PropertyBinding* property.

The following formatting options are supports for the label text:

- WrapText
- HorizontalAlignment
- VerticalAlignment
- DirectionRightToLeft
- DirectionVertical
- TextCase
- NoClip
- LineLimit
- FitBlackBox
- MeasureTrailingSpace

The following options are supports for customizing the appearance of the label.

- BackGroundStyle
- FontColorStyle
- FontStyle

Use Case Scenarios

When you are drawing a Business Process Flow Diagram, using this support, you can name the node representing the stage.

Tables for Properties and Methods

Properties

Table 1: Properies Table

Property	Description	Type	Data Type	Reference links

Name	Used to specify the name of a label.	NA	string	NA
FullName	Used to specify the full name of a label. Label container's name is prefixed with its name.	NA	string	NA
ReadOnly	Used to add a flag indicating whether the text object is Read-only or not.	NA	Boolean	NA
Visible	Specifies the visibility of the label.	NA	Boolean	NA
FontColorStyle	Specifies the font color style.	NA	FillStyle	NA
BackgroundStyle	Specifies the text background style.	NA	FillStyle	NA
Text	Gets or sets the label text.	NA	String	NA
FontStyle	Specifies the font style.	NA	FontStyle	NA
PropertyBinding	Binds the text value of the label to the Text property.	NA	LabelPropertyBinding	NA
Size	Gets or sets the label text size.	NA	SizeF	NA
SizeToNode	Set the node size to label size. This can be enabled if the node has only one label and label's position is	NA	Boolean	NA

	center.			
UpdatePosition	Gets or Sets whether default positioning has to be used.	NA	Boolean	NA
AdjustRotationAngle	Gets or Sets whether the label should remain horizontal on rotation of the node.	NA	Boolean	NA
TextCase	Gets or sets the case of the text in the label.	NA	TextCases	NA
HorizontalAlignment	Gets or sets the horizontal alignment of the text.	NA	StringAlignment	NA
VerticalAlignment	Gets or sets the vertical alignment of the text.	NA	StringAlignment	NA
FormatFlags	Gets the flags used to format the text.	NA	StringFormatFlags	NA
WrapText	Gets or sets a value indicating whether text should be wrapped, when it exceeds the width of the bounding box.	NA	Boolean	NA
DirectionRightToLeft	Gets or sets a value indicating whether the text is right to left.	NA	Boolean	NA
DirectionVertical	Gets or sets a value indicating whether the text is vertical.	NA	Boolean	NA

FitBlackBox	Gets or sets a value indicating whether no part of any glyph overhangs the label bounds.	NA	Boolean	NA
LineLimit	Gets or sets a value indicating whether only entire lines are laid out in the formatting rectangle.	NA	Boolean	NA
MeasureTrailingSpaces	Gets or sets a value indicating whether space at the end of each line in calculations that measure the size of the text.	NA	Boolean	NA
NoClip	Gets or sets a value indicating whether overhanging parts of glyphs and unwrapped text reaching outside the formatting rectangle are allowed to show.	NA	Boolean	NA
OffsetX	Gets or sets the label offset x in percent relative to node's width from node's top left point.	NA	float	NA
OffsetY	Gets or sets the label offset y in percent relative to node's width from node's top left point.	NA	float	NA

Position	Gets or sets the position of the label.	NA	Position	NA

Methods

Table 2: Methods Table

Method	Description	Parameters	Type	Return Type	Reference links
GetPosition	Gets the label position in the node coordinates.	Empty	NA	PointF	NA
GetStringFormat	Creates a StringFormat object that encapsulates the properties of the text object.	Empty	NA	StringFormat	NA

Adding a Label to an Application

You can create label as illustrated in the following code example:

```
[C#]
RoundRect node = new RoundRect(0, 0, 170, 100,MeasureUnits.Pixel);

            //Create a label with predefined position
            Syncfusion.Windows.Forms.Diagram.Label lbl_TopCenter = new
Syncfusion.Windows.Forms.Diagram.Label(node, "Label_TopCenter");
            //Postion the label
            lbl_TopCenter.Position = Position.TopCenter;
            /* Position enum has the values Center, TopLeft, TopCenter,
TopRight, MiddleLeft, MiddleRight, BottomLeft, BottomCenter,
BottomRight and Custom */

            //Apply font style
            lbl_TopCenter.FontStyle.Bold = true;
            lbl_TopCenter.FontStyle.Family = "Corbel";
            lbl_TopCenter.FontStyle.Size = 9;
            //Add the label to node's label collection
            node.Labels.Add(lbl_TopCenter);
```

```
            //Create a lable with custom position
            Syncfusion.Windows.Forms.Diagram.Label lbl_Custom = new
Syncfusion.Windows.Forms.Diagram.Label(node, "Label_Custom");
            //Postion the label
            lbl_Custom.Position = Position.Custom;
            lbl_Custom.OffsetX = 0;
            lbl_Custom.OffsetY = 0;
            //Apply font style
            lbl_Custom.FontStyle.Bold = true;
            lbl_Custom.FontStyle.Family = "Corbel";
            lbl_Custom.FontStyle.Size = 9;
            //Format the label text
            lbl_Custom.HorizontalAlignment = StringAlignment.Center;
            lbl_Custom.VerticalAlignment = StringAlignment.Far;
            //WrapText is set to true by default

            //Add the label to node's label collection
            node.Labels.Add(lbl_Custom);
            //Add the node to diagram model
            diagram1.Model.AppendChild(node);
```

[VB]

```
Dim node As New RoundRect(0, 0, 170, 100,MeasureUnits.Pixel)

                'Create a label with predefined position
                Dim lbl_TopCenter As New
Syncfusion.Windows.Forms.Diagram.Label(node, "Label_TopCenter")
                'Postion the label
                lbl_TopCenter.Position = Position.TopCenter
                ' Position enum has the values Center, TopLeft,
TopCenter, TopRight, MiddleLeft, MiddleRight, BottomLeft, BottomCenter,
BottomRight and Custom

                'Apply font style
                lbl_TopCenter.FontStyle.Bold = True
                lbl_TopCenter.FontStyle.Family = "Corbel"
                lbl_TopCenter.FontStyle.Size = 9
                'Add the label to node's label collection
                node.Labels.Add(lbl_TopCenter)

                'Create a lable with custom position
                Dim lbl_Custom As New
Syncfusion.Windows.Forms.Diagram.Label(node, "Label_Custom")
                'Postion the label
                lbl_Custom.Position = Position.Custom
                lbl_Custom.OffsetX = 0
                lbl_Custom.OffsetY = 0
                'Apply font style
                lbl_Custom.FontStyle.Bold = True
                lbl_Custom.FontStyle.Family = "Corbel"
                lbl_Custom.FontStyle.Size = 9
```



```
                    'Format the label text
                    lbl_Custom.HorizontalAlignment =
StringAlignment.Center
                    lbl_Custom.VerticalAlignment = StringAlignment.Far
                    'WrapText is true by default

                    'Add the label to node's label collection
                    node.Labels.Add(lbl_Custom)
                    'Add the node to diagram model
                    diagram1.Model.AppendChild(node)
```

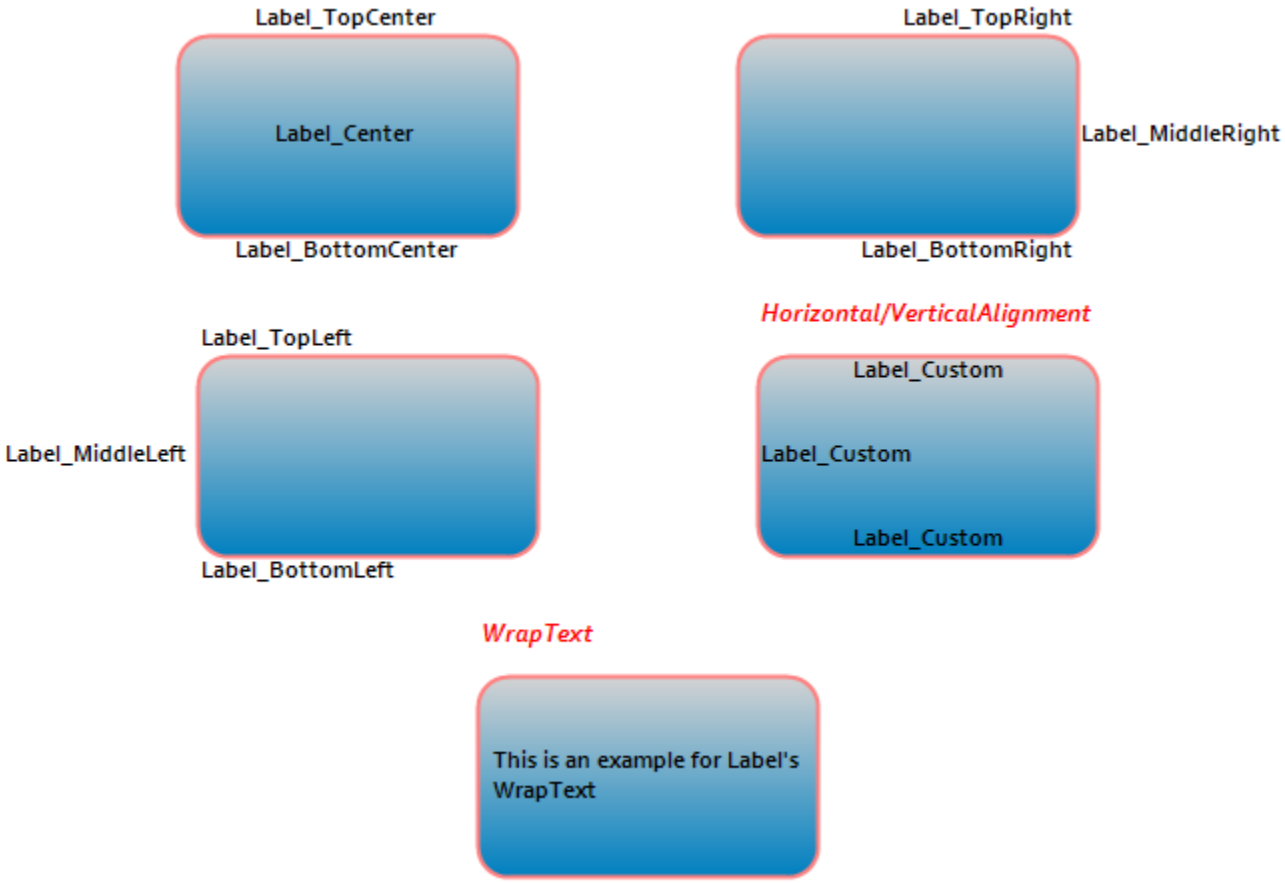

Figure 72: Label

4.4.5 Label Orientation

This feature lets the user to specify the orientation of a node label. There are two orientation modes, Horizontal, which displays the label horizontally, and Auto, which rotates the label based on the node or connector rotation angle.

Properties

Property	Description	Data Type
Orientation	Gets or sets the orientation of the label.	Enum

The following code shows how to set the orientation of the label to horizontal:

[C#]

```
Syncfusion.Windows.Forms.Diagram.Label label = new
Syncfusion.Windows.Forms.Diagram.Label(node, "Orientation");
//Sets the orientation of the label as horizontal.
label.Orientation = LabelOrientation.Horizontal;
node.Labels.Add(label);
```

[VB]

```
Dim label As New Syncfusion.Windows.Forms.Diagram.Label(node,
"Orientation")
'Sets the orientation of the label as horizontal.
label.Orientation = LabelOrientation.Horizontal
node.Labels.Add(label)
```

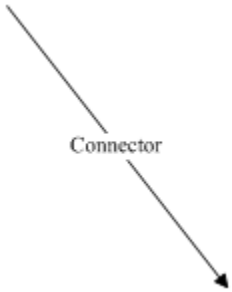

Figure 73: Label with Horizontal orientation

Figure 74: Label with Automatic orientation

4.5 Layout Management

Different Layout Manager are discussed under the following section:

4.5.1 Manual Layout

The various types of Layout Managers that are supported by the diagram control to align nodes are as follows.

Some common Layout Manager settings are discussed in the below topic.

- Layout Manager Settings

4.5.1.1 Table Layout Manager

The **TableLayoutManager** class can be used to arrange various objects in columns and rows in a table format. The **TableTreeLayoutManager** arranges nodes in a Table layout, positioning the nodes in a rectangular grid of cells, with each node spanning over a single table cell. The TableTreeLayoutManager is used when tabular relationships need to be depicted. The various properties of the TableTreeLayoutManager are listed below.

The model and the number of rows and column values are passed as parameters to the TableLayoutManager class. The parameters and properties involved with the TableLayoutManager are listed in the below table.

Property	Description
Model	The Model to be attached to the Layout Manager.
CellSizeMode	Gets / sets the cell size mode with one of the

	following options: • EqualToMaxNode • MinimalTable • Minimal
MaxSize	Gets / sets the size of each table cell. It is an integer type value.
MaxColumnCount	Represents the maximum horizontal cell count in the table. It is an integer type value.
MaxRowsCount	Represents the maximum vertical cell count in the table. It is an integer type value.
HorizontalSpacing	Defines the horizontal offset between adjacent nodes.
VerticalSpacing	Defines the vertical offset between adjacent nodes.

Programmatically, the table layout manager instance should be created with the respective arguments, assigned to the Layout Manager and updated as follows.

[C#]

```
TableLayoutManagermt lLayout=new
TableLayoutManager(this.diagram1.Model, 7, 7);
tlLayout.VerticalSpacing = 20;
tlLayout.HorizontalSpacing = 20;
tlLayout.CellSizeMode = CellSizeMode.EqualToMaxNode;
tlLayout.Orientation = Orientation.Horizontal;
tlLayout.MaxSize = new SizeF(500, 600);

this.diagram1.LayoutManager = tlLayout;
this.diagram1.LayoutManager.UpdateLayout(null);.AttachModel(model1);
documentExplorer1.Dock = DockStyle.Right;
documentExplorer1.BackColor = System.Drawing.SystemColors.Window;
documentExplorer1.Location = new System.Drawing.Point(0, 377);
documentExplorer1.Size = new System.Drawing.Size(200, 100);
documentExplorer1.BorderStyle =
System.Windows.Forms.BorderStyle.Fixed3D;
documentExplorer1.ShowNodeToolTips = true;
```

[VB]

```
Dim lLayout As TableLayoutManagermt = New
TableLayoutManager(Me.diagram1.Model, 7, 7)
tlLayout.VerticalSpacing = 20
tlLayout.HorizontalSpacing = 20
tlLayout.CellSizeMode = CellSizeMode.EqualToMaxNode
tlLayout.Orientation = Orientation.Horizontal
tlLayout.MaxSize = New SizeF(500, 600)

Me.diagram1.LayoutManager = tlLayout
Me.diagram1.LayoutManager.UpdateLayout(Nothing).AttachModel(model1)
documentExplorer1.Dock = DockStyle.Right
documentExplorer1.BackColor = System.Drawing.SystemColors.Window
documentExplorer1.Location = New System.Drawing.Point(0, 377)
documentExplorer1.Size = New System.Drawing.Size(200, 100)
documentExplorer1.BorderStyle =
System.Windows.Forms.BorderStyle.Fixed3D
documentExplorer1.ShowNodeToolTips = True
```

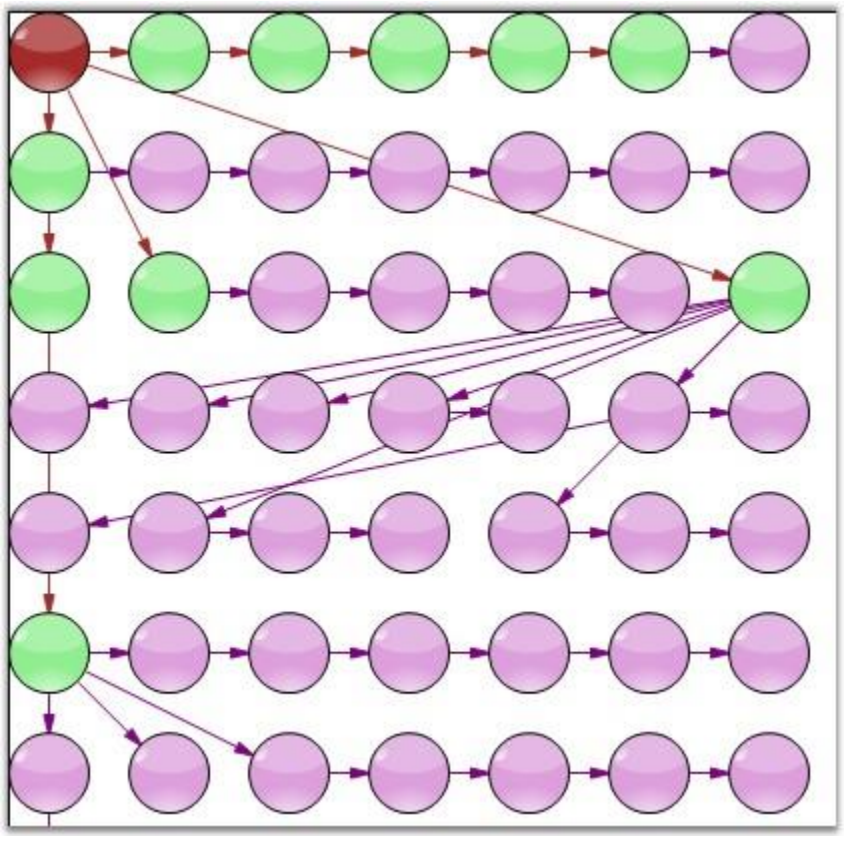

Figure 75: Horizontal Orientation

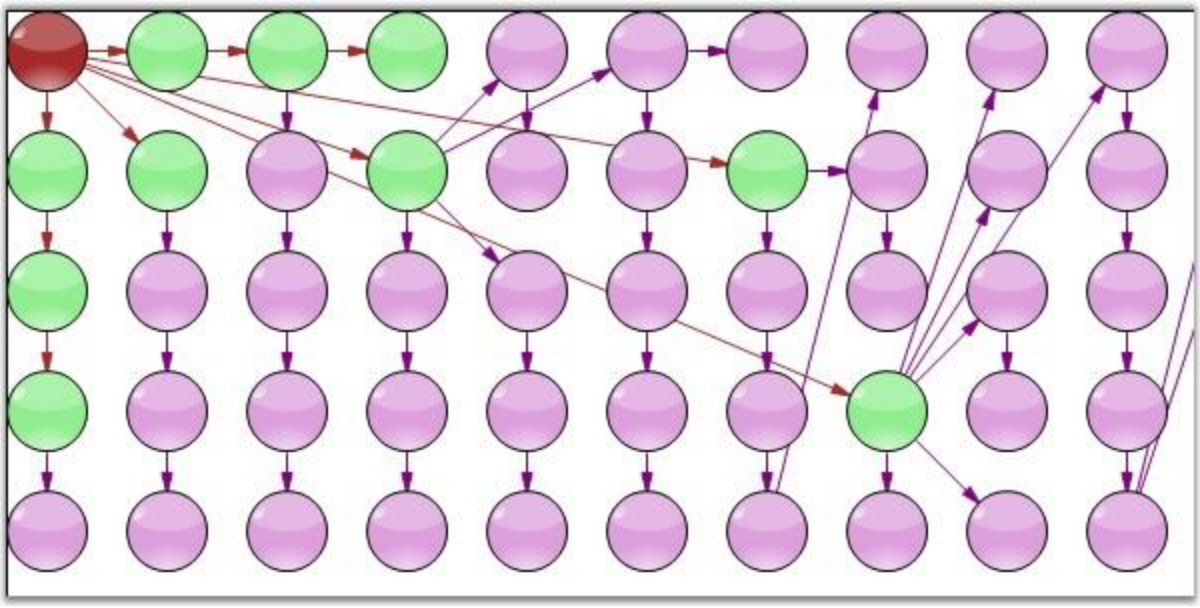

Figure 76: Vertical Orientation

4.5.1.2 Directed Tree Layout Manager

The Directed Tree Layout Manager implements a layout manager for arranging nodes in a tree-like structure. This Layout manager can be applied to any diagram that is composed of a directed tree graph with unique root and child nodes. The layout manager lets you orient the tree in any direction around the root and can be used for creating arrangements such as top-to-bottom vertical trees, bottom-to-top vertical trees, right-to-left horizontal trees, left-to-right horizontal trees, angular trees etc.

The **DirectedTreeLayoutManager** class is a subclass of the **GraphLayoutManager** and implements a layout manager for arranging nodes in a tree-like structure. The tree layout can be applied to any diagram that is composed as a directed tree graph with a unique root and child nodes. The layout manager lets you orient the tree in just about any direction around the root and can be used for creating arrangements such as top-to-bottom vertical trees, bottom-to-top vertical trees, right-to-left horizontal trees, left-to-right horizontal trees, angular trees and so on.

Graph orientation is determined by the rotation degree parameter while initializing the layout manager. A rotation degree of 0° specifies a top-to-bottom vertical tree while a rotation degree of 270° will result in a left-to-right horizontal tree layout.

The parameters to be defined for the DirectedTreeLayoutManager class are listed in the below table.

Property	Description
Model	Represents the model of the diagram, which has to be displayed out as a directed tree.
RotationAngle	Gets or sets the rotation angle for the graph. It accepts only integer values between 0-360.
HorizontalSpacing	Holds the value for the horizontal offset between adjacent nodes (float value).
VerticalSpacing	Holds the value for the vertical offset between adjacent nodes (float value).

Programmatically, the directed tree layout manager instance is created with the respective arguments, assigned to the Layout Manager and updated as follows.

[C#]

```
DirectedTreeLayoutManager directedLayout = new
DirectedTreeLayoutManager(diagram1.Model,0, 20, 20);
diagram1.LayoutManager = directedLayout;
diagram1.LayoutManager.UpdateLayout(null);
```

[VB]

```
DirectedTreeLayoutManager directedLayout = new
DirectedTreeLayoutManager(diagram1.Model,0, 20, 20);
diagram1.LayoutManager = directedLayout;
diagram1.LayoutManager.UpdateLayout(null);
```

Sample Diagrams are as follows.

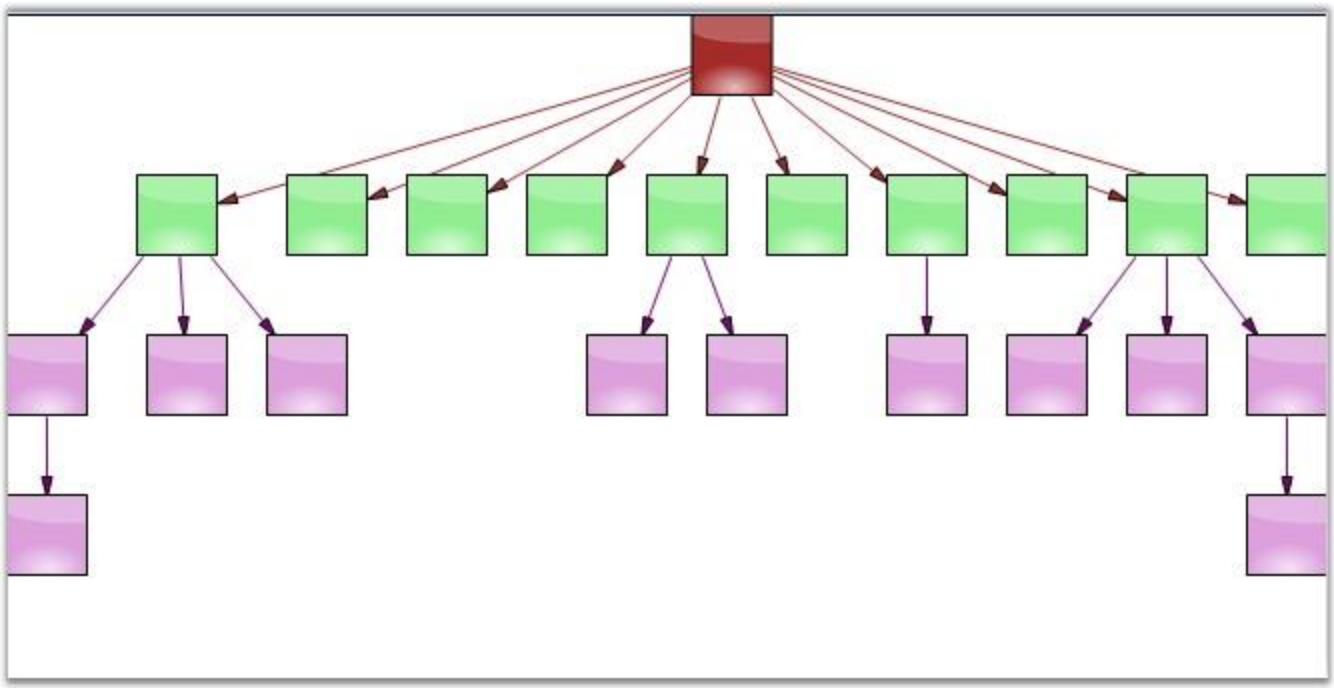

Figure 77: Top-to-Bottom Directed Tree Layout with 0 degree Rotation Angle

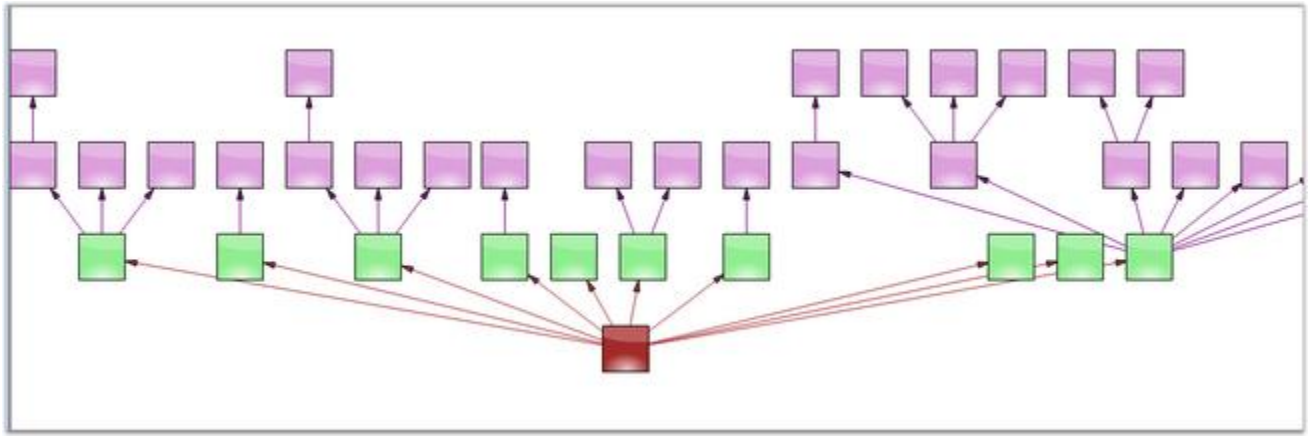

Figure 78: Bottom-to-Top Directed Tree Layout with 180 degree Rotation Angle

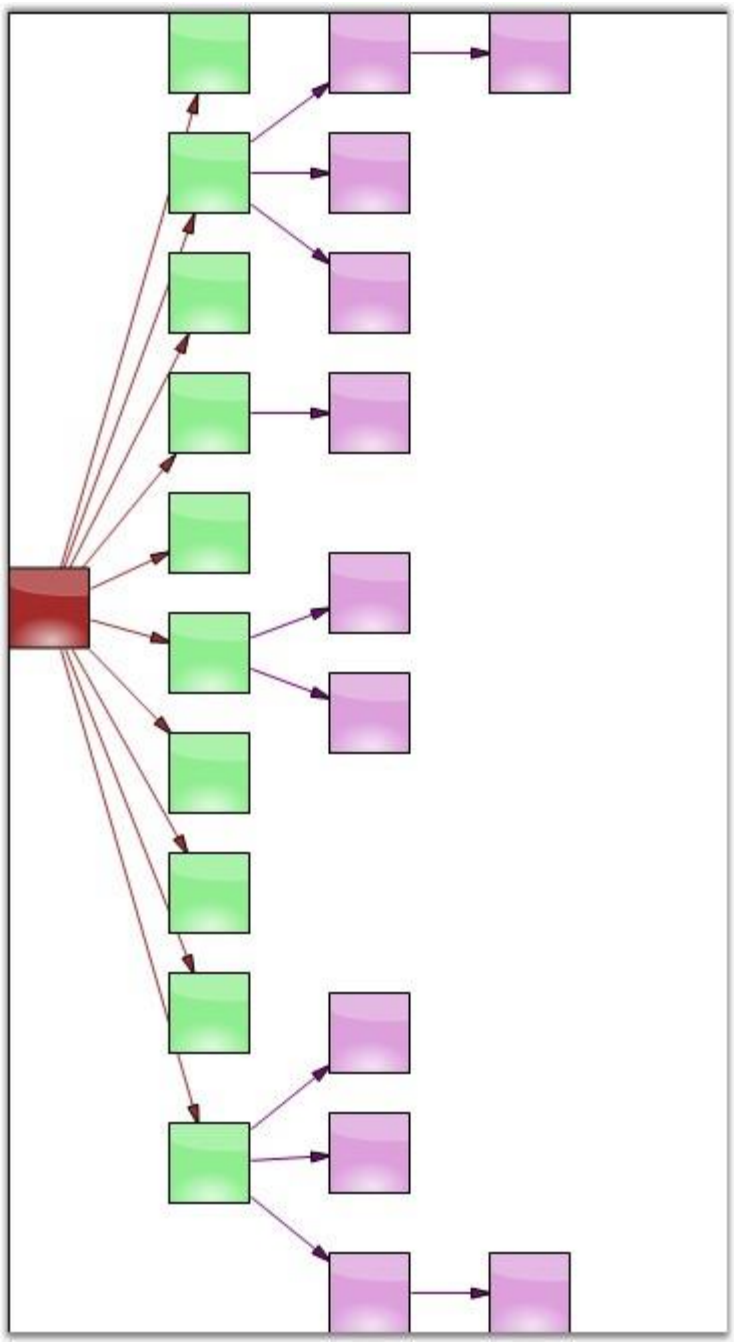

Figure 79: Left-to-Right with 270 degree Rotation Angle

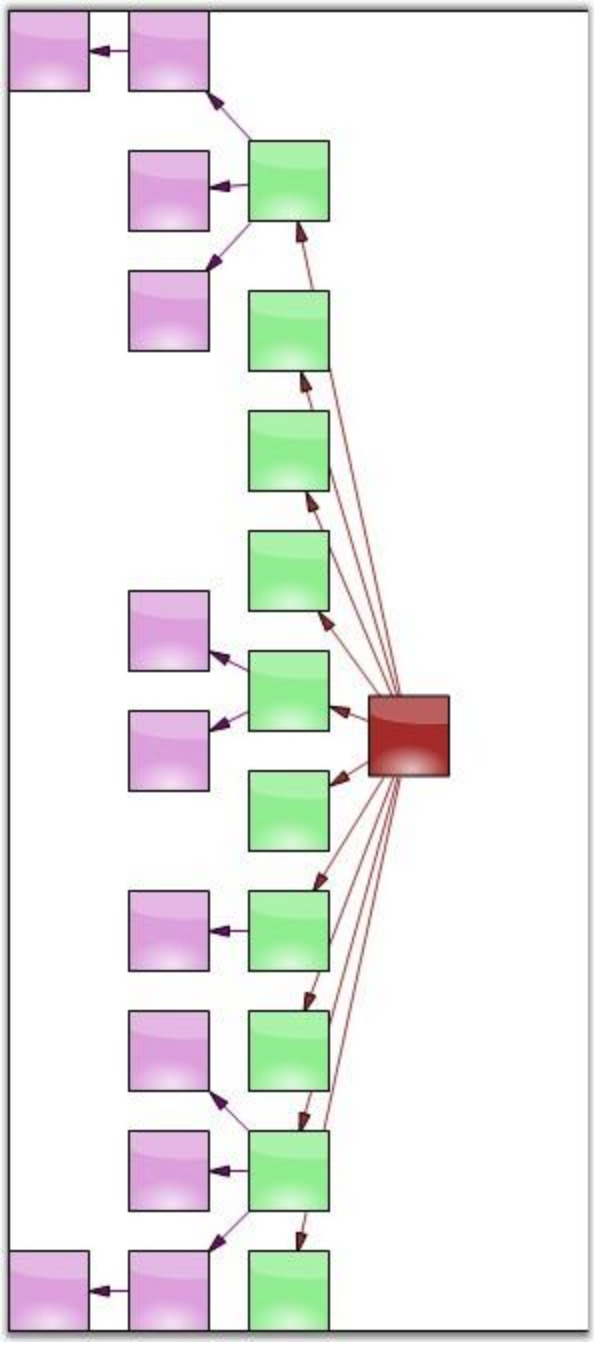

Figure 80: Right-to- Left with 90 degree Rotation Angle

4.5.1.3 Radial Tree Layout Manager

The Radial Tree Layout Manager is a specialization of the Directed Tree Layout Manager that employs a circular layout algorithm for locating the diagram nodes. The RadialTreeLayoutManager arranges nodes in a circular layout, positioning the root node at the center of the graph and the child nodes in a circular fashion around the root. Sub-trees formed by the branching of child nodes are located radially around the child nodes. This arrangement results in an ever-expanding concentric arrangement with radial proximity to the root node indicating the node level in the hierarchy.

The following parameters need should be specified for the RadialTreeLayoutManager.

Property	Description
Model	Represents the model to be attached to the Layout Manager.
RotationAngle	Defines the Graph Rotation angle. It accepts only integer values between 0-360.
HorizontalSpacing	Defines the horizontal offset between adjacent nodes.
VerticalSpacing	Defines the vertical offset between adjacent nodes.

Programmatically, the radial tree layout manager instance is created with the respective arguments, assigned to the Layout Manager and updated as follows:

[C#]

```
RadialTreeLayoutManager radialLayout= new
RadialTreeLayoutManager(model1, 0, 20, 20);
this.diagram1.LayoutManager=radialLayout;
this.diagram1.LayoutManager.UpdateLayout(null);
```

[VB]

```
Dim radialLayout As New RadialTreeLayoutManager(model1, 0, 20, 20)
Me.diagram1.LayoutManager = radialLayout
Me.diagram1.LayoutManager.UpdateLayout(Nothing)
```

Sample Diagram is as follows.

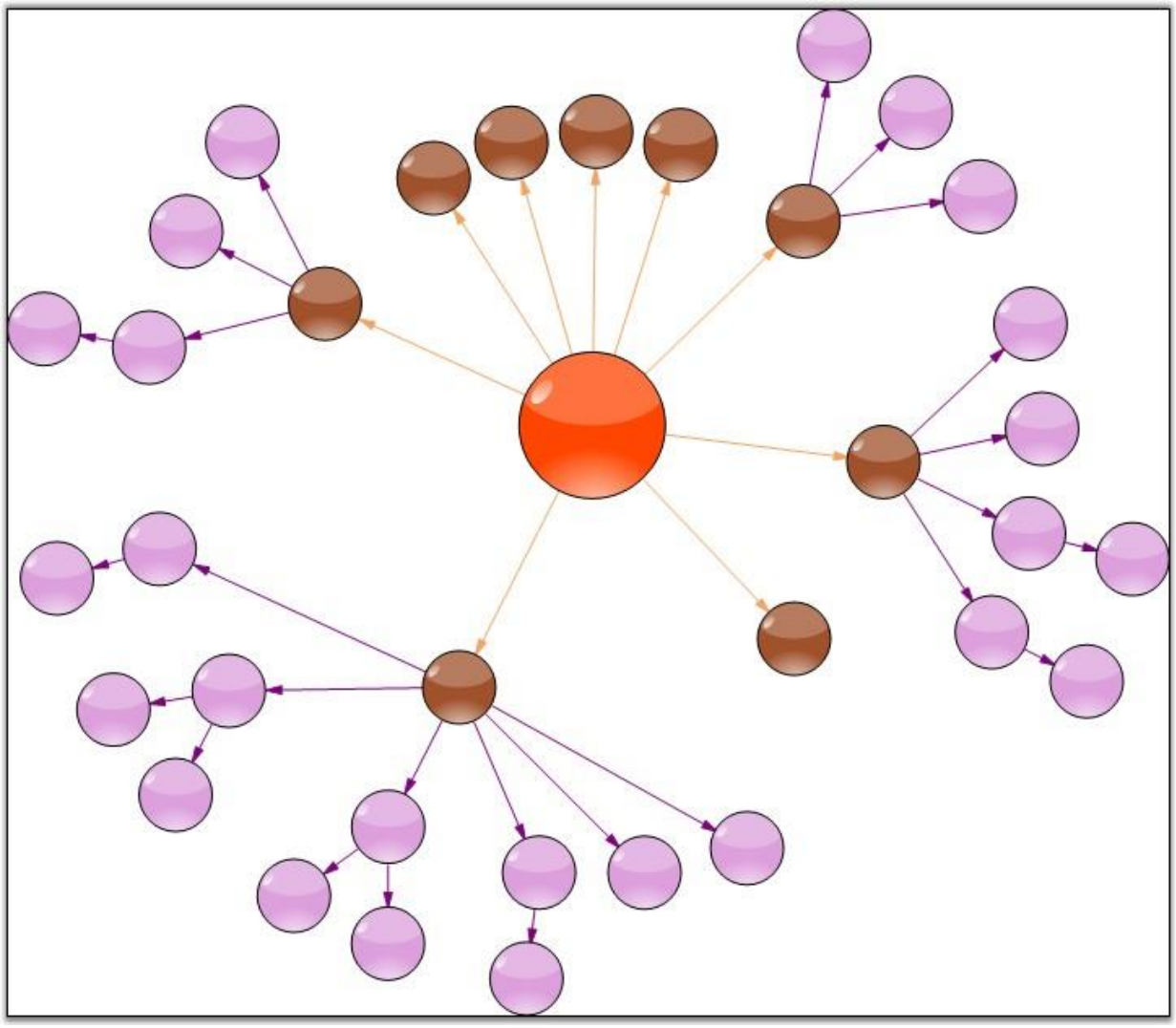

Figure 81: Radial Tree Layout with 0 degree Rotation Angle

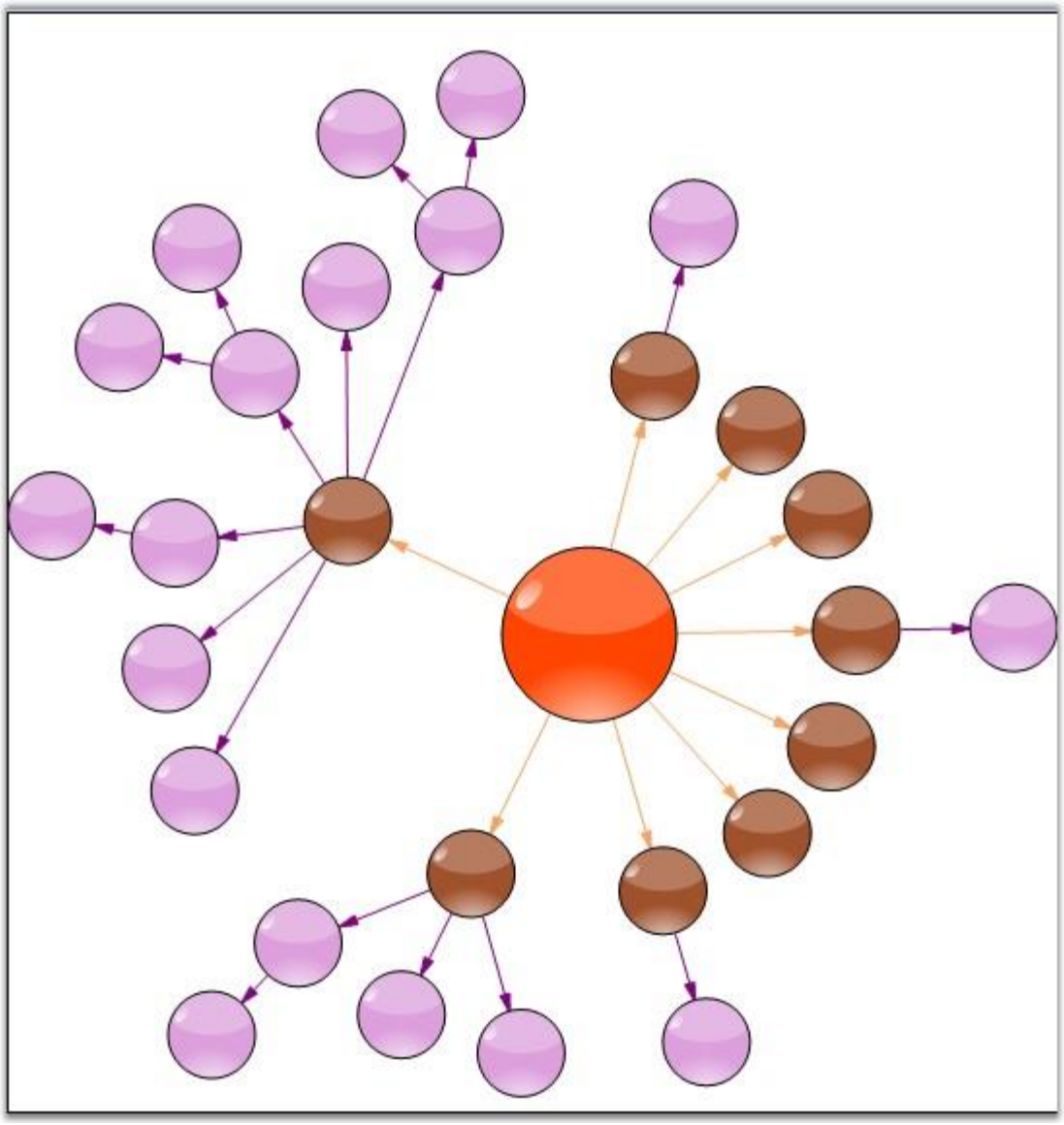

Figure 82: Radial Tree Layout with 180 degree Rotation Angle

4.5.1.4 Symmetric Layout Manager

The Symmetric layout manager arranges all the nodes in a symmetric fashion using the vertical input distance, which specifies the distance between the nodes.

The Model and Vertical Distance values are passed as parameters to the **SymmetricLayoutManager** class. The parameters and properties of Symmetric Layout Manager is listed below.

Property	Description
Model	Represents the model of the diagram, which has to be displayed out as a directed tree.
VerticalDistance	Defines the Graph Rotation angle. It accepts only integer values between 0 - 360.
SpringFactor	Gets or sets the spring factor.
SpringLength	Defines the spring length.
MaxIteration	Holds the maximum count of iteration.

Programmatically, the symmetric layout manager instance is created with the respective arguments, assigned to the LayoutManager and updated as follows.

[C#]

```
SymmetricLayoutManager symmetricLayout = new
SymmetricLayoutManager(diagram1.Model,100);
symmetricLayout.SpringFactor = 0.442;
symmetricLayout.SpringLength = 100;
symmetricLayout.MaxIteration = 500;
this.diagram1.LayoutManager = symmetricLayout;
this.diagram1.LayoutManager.UpdateLayout(null);
```

[VB]

```
Dim symmetricLayout As New SymmetricLayoutManager(diagram1.Model, 100)
symmetricLayout.SpringFactor = 0.442
symmetricLayout.SpringLength = 100
symmetricLayout.MaxIteration = 500
Me.diagram1.LayoutManager = symmetricLayout
Me.diagram1.LayoutManager.UpdateLayout(Nothing)
```

Sample Diagrams are as follows.

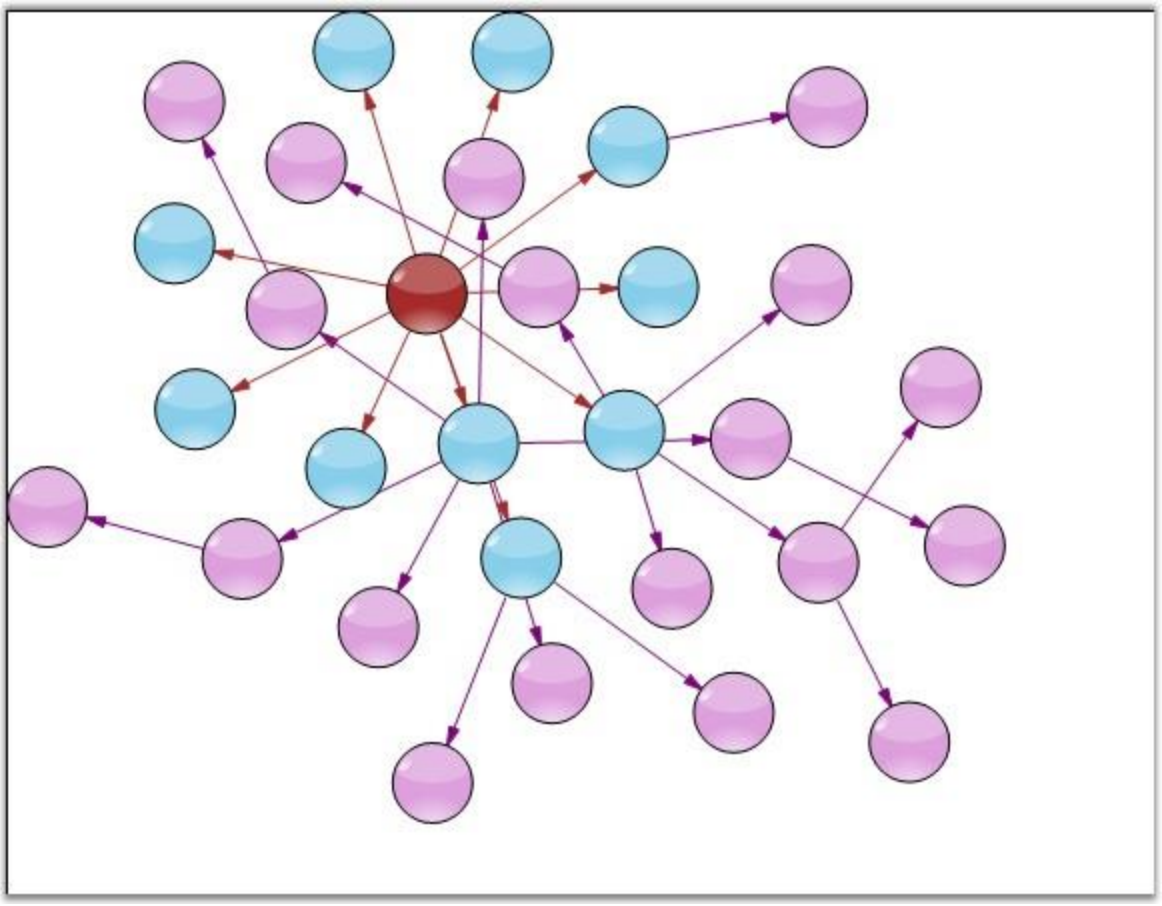

Figure 83: Diagram With Symmetric Layout

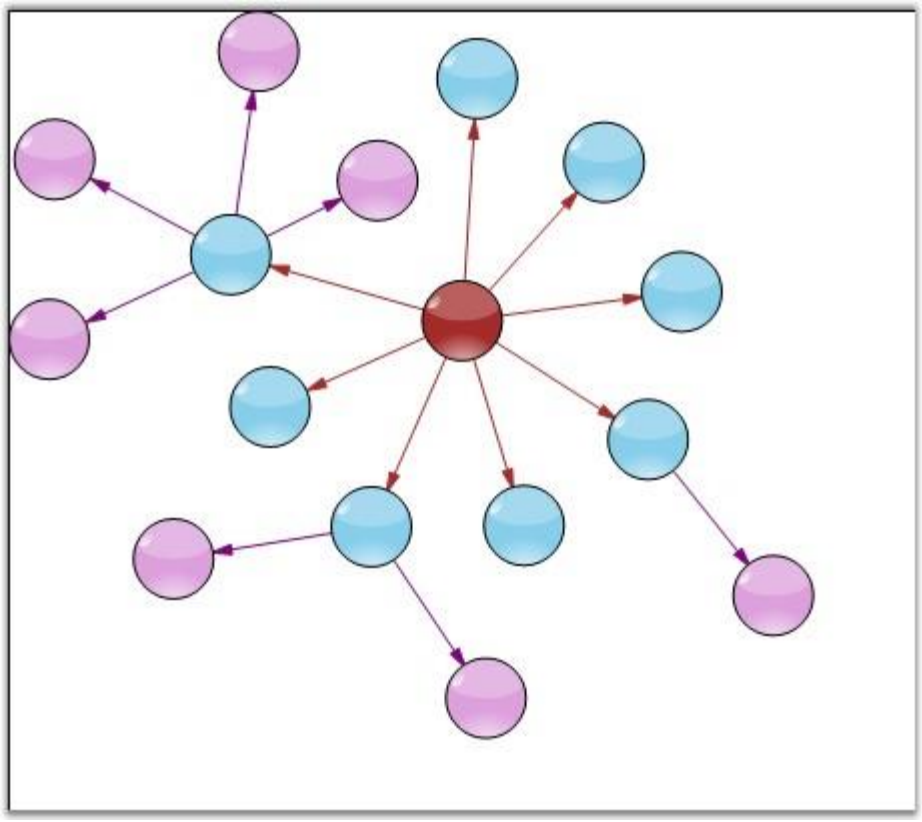

Figure 84: Symmetric Layout with Spring Factor Settings

4.5.1.5 Hierarchical Layout Manager

Hierarchical layout manager arranges the nodes in a hierarchical fashion depending on the parent-child relationship. Unlike the directed layout, more than one parent item can be defined for a child.

The parameters to be passed for the **HierarchicalLayoutManager** class are as follows:

Property	Description
Model	Represents the model of the diagram, which has to be displayed out as a directed tree.
RotationAngle	Defines the Graph Rotation angle. It accepts only integer values between 0-360.

HorizontalSpacing	Holds the value for the horizontal offset between adjacent nodes (float value).
VerticalSpacing	Holds the value for the vertical offset between adjacent nodes (float value).

Programmatically, the hierarchical layout manager instance should be created with the respective arguments, assigned to the Layout Manager and updated as follows.

```
[C#]

HierarchicLayoutManager hierarchyLayout = new HierarchicLayoutManager
(diagram1.Model, 0, 10, 20);
this.diagram1.LayoutManager = hierarchyLayout;
this.diagram1.LayoutManager.UpdateLayout(null);
```

```
[VB]

Dim hierarchyLayout As New HierarchicLayoutManager(diagram1.Model, 0,
10, 20)
Me.diagram1.LayoutManager = hierarchyLayout
Me.diagram1.LayoutManager.UpdateLayout(Nothing)
```

Sample diagrams are as follows:

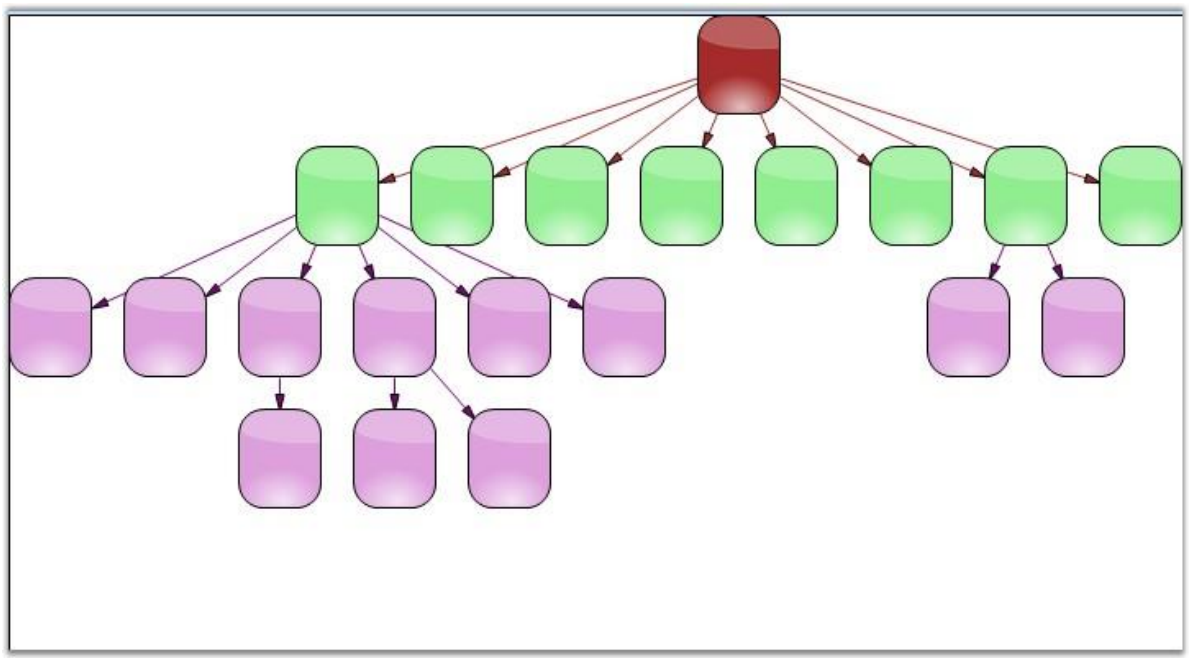

Figure 85: Top-to-Bottom Hierarchical Tree Layout

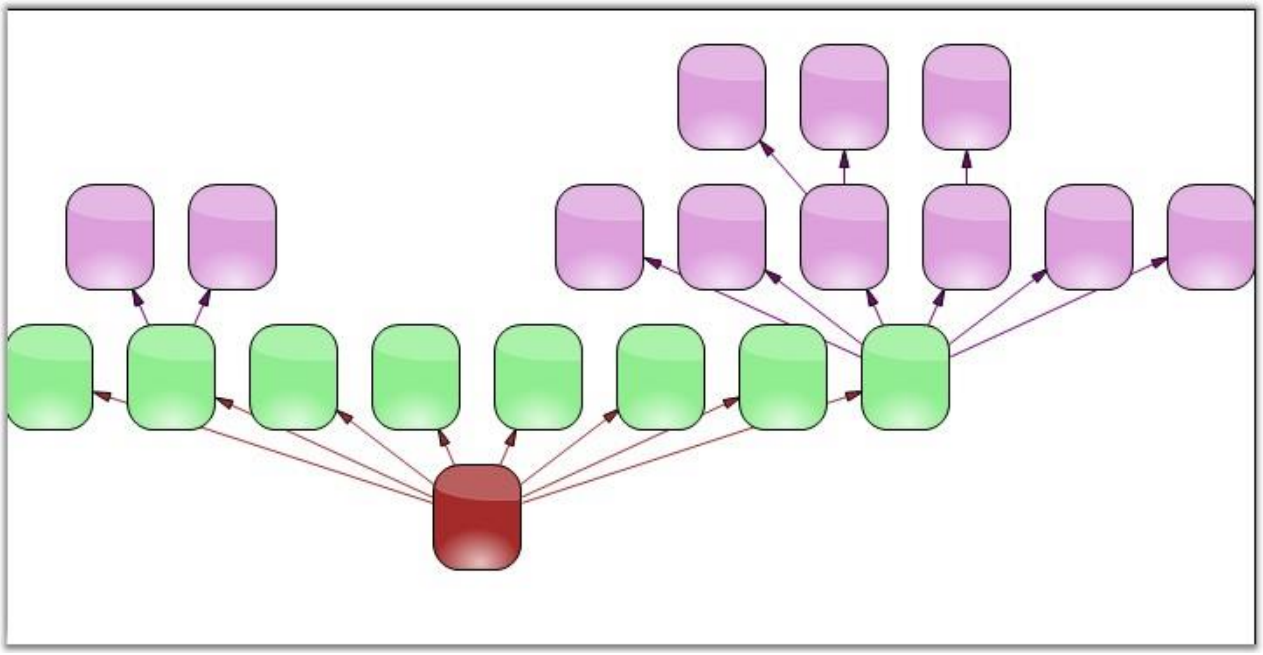

Figure 86: Bottom-to-Top Hierarchical Tree Layout

4.5.1.6 Graph Layout Manager

The **GraphLayoutManager** is an abstract base class that can be used for implementing layout managers for diagrams composed primarily of nodes forming connected graphs. The GraphLayoutManager implements the infrastructure for initializing, validating and creating the diagram graph by enumerating the diagram model's child nodes. It also enables positioning diagram nodes using the layout strategies provided by specialized directed tree layout managers that derive from it.

The event of the Graph Layout Manager class and its description is given below:

Event	Description
PreferredLayout Event	Event provides the application a chance to customize the layout of the diagram.

Programmatically, it is implemented as follows.

[C#]

```
RadialTreeLayoutManager dtlm = new
RadialTreeLayoutManager(this.diagram1.Model, 0, 20, 20);
dtlm.PreferredLayout += new
PreferredLayoutEventHandler(dtlm_PreferredLayout);
this.diagram1.LayoutManager = dtlm;
this.diagram1.LayoutManager.UpdateLayout(null);
this.diagram1.UpdateView();

private void dtlm_PreferredLayout(object sender,
PreferredLayoutEventArgs evtArgs)
 {
       if (evtArgs.IsGraphUnderLayout)
       {
            evtArgs.ResizeGraphNodes = false;
            evtArgs.Location = new PointF(150,150);
            evtArgs.Size = new SizeF(100, 100);
       }
 }
```

[VB]

```
Dim dtlm As New RadialTreeLayoutManager(Me.diagram1.Model, 0, 20, 20)
AddHandler dtlm.PreferredLayout, AddressOf dtlm_PreferredLayout
Me.diagram1.LayoutManager = dtlm
Me.diagram1.LayoutManager.UpdateLayout(Nothing)
Me.diagram1.UpdateView()

Private Sub dtlm_PreferredLayout(ByVal sender As Object, ByVal evtArgs
As PreferredLayoutEventArgs)
    If evtArgs.IsGraphUnderLayout Then
        evtArgs.ResizeGraphNodes = False
        evtArgs.Location = New PointF(150, 150)
        evtArgs.Size = New SizeF(100, 100)
    End If
End Sub
```

The following is a sample Diagram showing GraphLayoutManager:

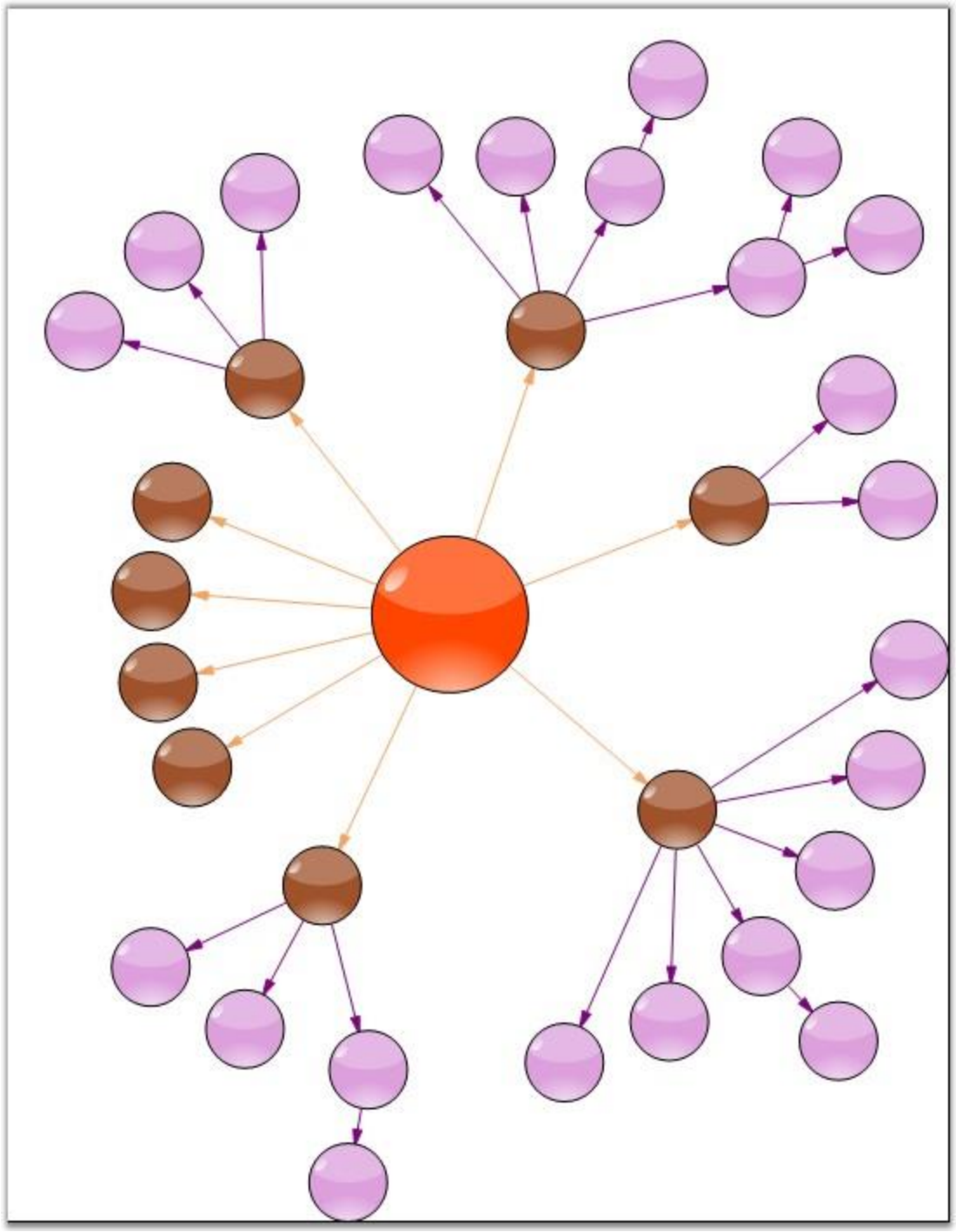

Figure 87: Graph Layout Manager

4.5.1.7 Subgraph Layout Manager

The SubgraphTreeLayoutManager enables the sub nodes of a diagram layout tree to have an orientation that is distinct from the parent node. The subgraph orientation is specified using a **SubgraphPreferredLayout** event that the layout manager raises before positioning each set of sub nodes in the graph.

The event of the SubgraphLayoutManager class is as follows:

Event	Description
SubgraphPreferredLayout	Event that the layout manager raises before positioning each set of sub nodes in the graph.

Programmatically, it is implemented as follows.

[C#]

```
SubgraphTreeLayoutManager st = new
SubgraphTreeLayoutManager(this.diagram1.Model,0, 20, 20);
st.SubgraphPreferredLayout += new SubgraphPreferredLayoutEventHandler
                  (st_SubgraphPreferredLayout);
this.diagram1.LayoutManager = st;
this.diagram1.LayoutManager.UpdateLayout(null);
this.diagram1.UpdateView();

private void st_SubgraphPreferredLayout(object sender,
SubgraphPreferredLayoutEventArgs evtArgs)
{
      evtArgs.ResizeSubgraphNodes=false;
      evtArgs.RotationDegree=0;
}
```

[VB]

```
Dim st As New SubgraphTreeLayoutManager(Me.diagram1.Model, 0, 20, 20)
AddHandler st.SubgraphPreferredLayout, AddressOf
st_SubgraphPreferredLayout
Me.diagram1.LayoutManager = st
Me.diagram1.LayoutManager.UpdateLayout(Nothing)
Me.diagram1.UpdateView()

Private Sub st_SubgraphPreferredLayout(ByVal sender As Object, ByVal
evtArgs As SubgraphPreferredLayoutEventArgs)
    evtArgs.ResizeSubgraphNodes = False
```

```
    evtArgs.RotationDegree = 0
End Sub
```

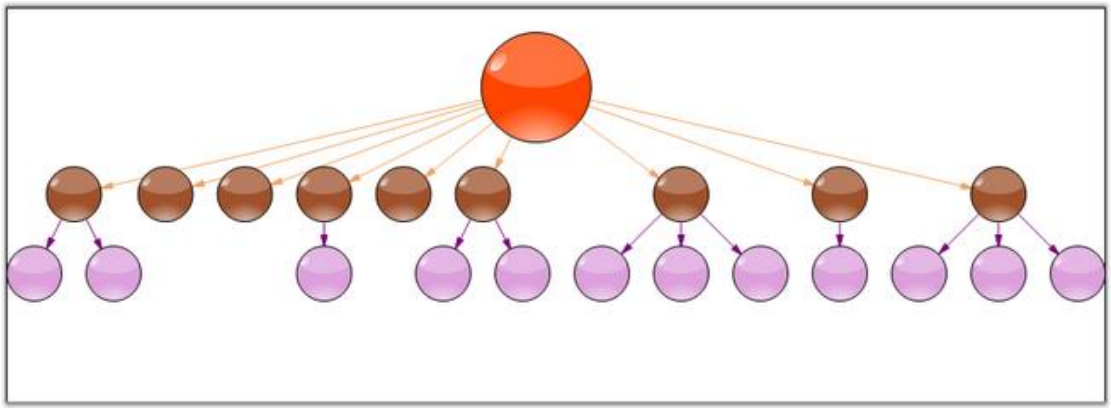

Figure 88: Sub Graph Tree Layout

4.5.1.8 OrgChart Layout Manager

Event arranges all the nodes in parent-child relationship with the new OrgLineConnector that connects the nodes to get the OrgLayout appearance. The OrgLineConnector is specially designed for connecting the nodes in OrgChartLayoutManager.

Property	Description
Model	Represents the model of the diagram, which is displayed as an OrgLayout.
RotationDirection	Gets / sets the layout directions. There are four major directions, which are as follows: • BottomToTop • LeftToRight • RightToLeft • TopToBottom
HorizontalSpacing	Holds the value for the horizontal offset between adjacent nodes (float value).
VerticalSpacing	Holds the value for the vertical offset between adjacent nodes (float value).

Programmatically, the organization layout manager instance should be created with the respective arguments, assigned to the Layout Manager and updated as follows.

```
[C#]

OrgChartLayoutManager manager = new
OrgChartLayoutManager(this.diagram.Model,RotateDirection.TopToBottom,
20, 50);
this.diagram1.LayoutManager = manager;
this.diagram1.LayoutManager.UpdateLayout(null);
```

```
[VB]

Dim manager As New OrgChartLayoutManager(Me.diagram.Model,
RotateDirection.TopToBottom, 20, 50)
Me.diagram1.LayoutManager = manager
Me.diagram1.LayoutManager.UpdateLayout(Nothing)
```

Sample diagram is as follows.

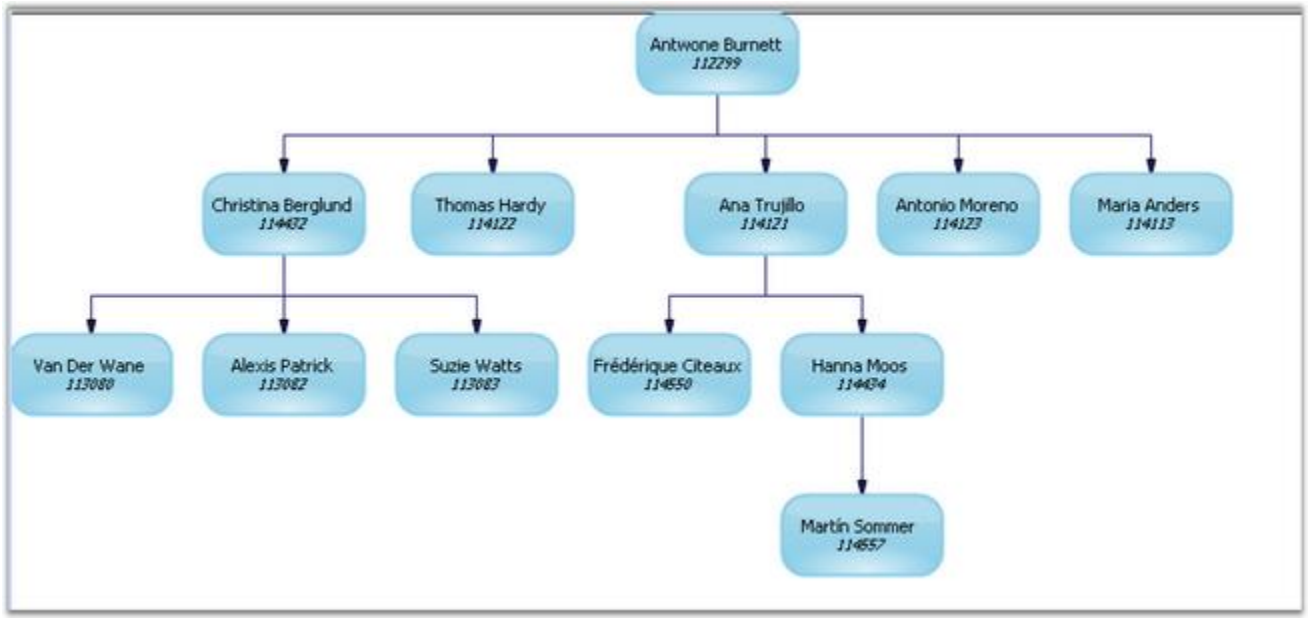

Figure 89: Top-to-Bottom Direction Organizational Chart Layout

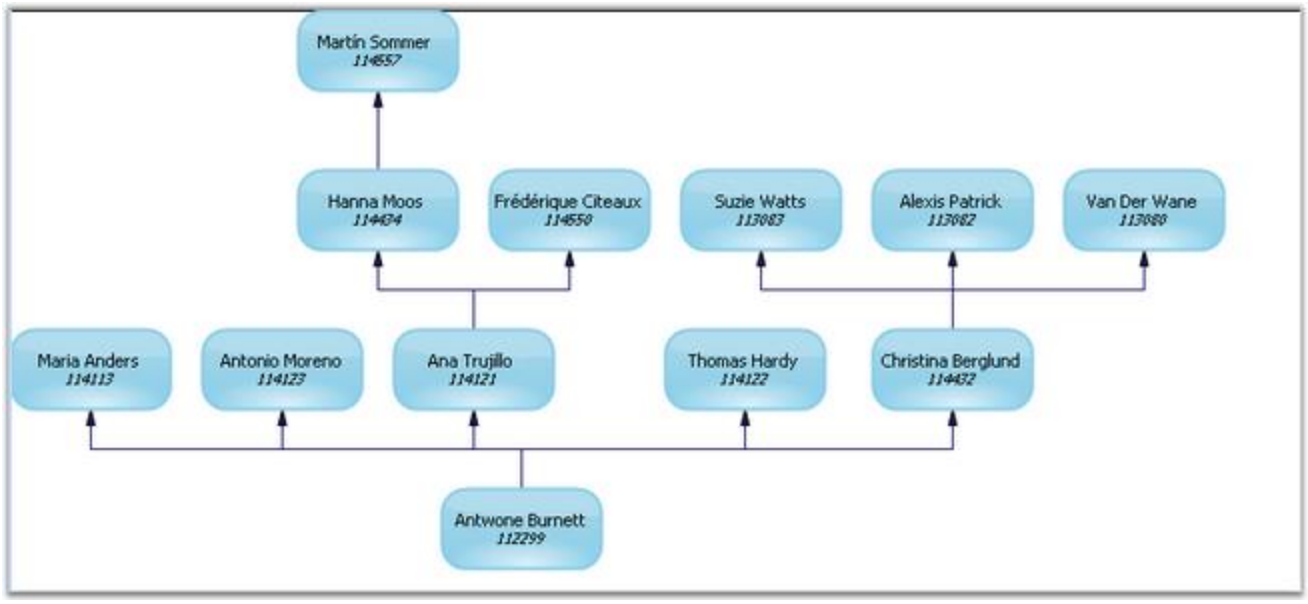

Figure 90: Bottom-to-Top Direction Organizational Chart Layout

OrgChart Alignment

As the OrgChartLayout follows a Waterfall model, whenever there is only one child node, the layout will be widened. To overcome this, Essential diagram enables you to align the single child node parallel to the parent node, which will reduce the layout structure.

```
[C#]

OrgChartLayoutManager manager = new
OrgChartLayoutManager(this.diagram.Model, RotateDirection.TopToBottom,
20, 50, LayoutType.Waterfall, 1, true);
```

```
[VB]

Dim manager as New
OrgChartLayoutManager(Me.diagram.Model,RotateDirection.TopToBottom, 20,
50,LayoutType.Waterfall, 1, True)
```

4.5.1.9 Layout Manager Settings

Margin Properties for Layout Managers

For all the Layout Managers supported by Essential Diagram, except ***Symmetric and Table Layout Manager***, it is now possible to set the left and right margins for the graph that can be laid out by the layout manager. The two new properties, i.e, **TopMargin** and **LeftMargin** of the Layout Manager will set the margin for the graph using the following code snippet.

```
[C#]

OrgChartLayoutManager manager = new
OrgChartLayoutManager(this.diagram.Model, RotateDirection.TopToBottom,
20, 50);
manager.LeftMargin = 50;
manager.TopMargin = 50;
```

```
[VB]

Dim manager as New
OrgChartLayoutManager(Me.diagram.Model,RotateDirection.TopToBottom, 20,
50)
manager.LeftMargin = 50
manager.TopMargin = 50
```

Improving performance

The performance of most of the Essential Diagram Layout Managers is now improved to a great extent. The time taken for laying out a diagram, using a Layout Manager can now be reduced by setting the **ImprovePerformance** property to ***true***.

```
[C#]

OrgChartLayoutManager manager = new
OrgChartLayoutManager(this.diagram.Model, RotateDirection.TopToBottom,
20, 50);
manager.ImprovePerformance = true;
```

```
[VB]

Dim manager as New
OrgChartLayoutManager(Me.diagram.Model,RotateDirection.TopToBottom, 20,
50)
manager.ImprovePerformance = True
```


4.6 Advanced Features

This particular feature section covers the below topics.

4.6.1 Node Selections

A node's behavior can be customized and modified using the EditStyle collection properties which can be used for the following:

- To prohibit selection, rotation and deletion of nodes, by using **AllowSelect**, **AllowRotate** and **AllowDelete** properties.
- To restrict a node's movement along the x or y axis, by using **AllowMoveX** and **AllowMoveY** properties.
- To prevent re-sizing the height and width of the node, by using **AllowChangeHeight** and **AllowChangeWidth** and **AllowResize** properties.

EditStyle Property	Description
AllowChangeHeight	Specifies whether or not to allow the height to be changed. Default value is ***True***.
AllowChangeWidth	Specifies whether or not to allow the width to be changed. Default value is ***True***.
AllowDelete	Specifies whether or not to allow the node to be deleted on clicking the DELETE key. Default value is ***True***.
AllowMoveX	Specifies whether or not to allow the node to be moved along the x-axis. Default value is ***True***.
AllowMoveY	Specifies whether or not to allow the node to be moved along the y-axis. Default value is ***True***.
AllowRotate	Specifies whether or not to rotate the node using the PinPoint. Default value is ***True***.
AllowSelect	Specifies whether or not to select the node on mouse click. Default value is ***True***.

Programmatically, the properties can be set as follows:

```
[C#]
```



```
rect.EditStyle.AllowChangeHeight = true;
rect.EditStyle.AllowChangeWidth = true;
rect.EditStyle.AllowDelete = false;
rect.EditStyle.AllowMoveX = true;
rect.EditStyle.AllowMoveY = false;
rect.EditStyle.AllowRotate = true;
rect.EditStyle.AllowSelect = true;
```

```
[VB]

rect.EditStyle.AllowChangeHeight = True
rect.EditStyle.AllowChangeWidth = True
rect.EditStyle.AllowDelete = False
rect.EditStyle.AllowMoveX = True
rect.EditStyle.AllowMoveY = False
rect.EditStyle.AllowRotate = True
rect.EditStyle.AllowSelect = True
```

In the above code snippets, the properties are set to the Rectangular node (rect) created through the code.

Behavior Settings

Property	Description
AspectRatio	Specifies whether to maintain the height and width ratio when the node is resized.
DefaultHandleEditMode	Specifies the mode in which the node should be handled. The default value for links and lines is Vertex and for all other nodes and polyline the default value is Resize. To move the nodes, DefaultHandleEditMode should be set to Resize. The options provided are as follows. • None • Resize • Vertex
Enabled	Specifies whether the node is enabled. Default value is ***True***.

AllowVertexEdit	Specifies whether or not to edit the vertex. Default value is ***True***.
HidePinPoint	Specifies whether to show or hide the PinPoint. Default value is ***False***.
HideRotationHandle	Specifies whether to show or hide the RotationHandle in order to control the rotation of the node. Default value is ***False***.

Programmatically these properties can be set as follows:

```
[C#]

rect.EditStyle.AspectRatio = true;
rect.EditStyle.DefaultHandleEditMode =HandleEditMode.Resize;
rect.EditStyle.Enabled = true;
rect.EditStyle.AllowVertexEdit = true;
rect.EditStyle.DefaultHandleEditMode =HandleEditMode.Vertex;
rect.EditStyle.HidePinPoint = true;
rect.EditStyle.HideRotationHandle = true;
```

```
[VB]

rect.EditStyle.AspectRatio = True
rect.EditStyle.DefaultHandleEditMode = HandleEditMode.Resize
rect.EditStyle.Enabled = True
rect.EditStyle.AllowVertexEdit = True
rect.EditStyle.DefaultHandleEditMode = HandleEditMode.Vertex
rect.EditStyle.HidePinPoint = True
rect.EditStyle.HideRotationHandle = True
```

In the above code snippets, the properties are set to the Rectangular node (rect) created through the code.

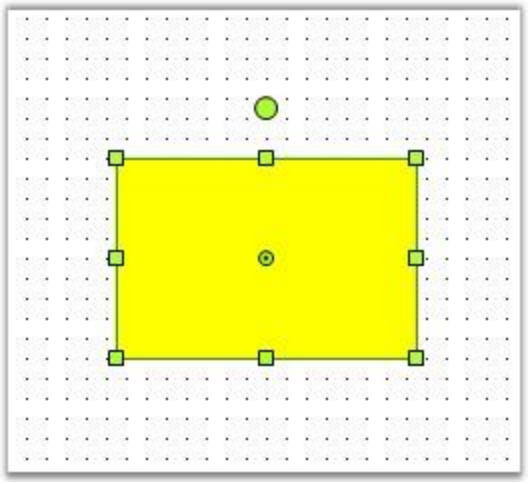

Figure 91: Default Handle Edit Mode

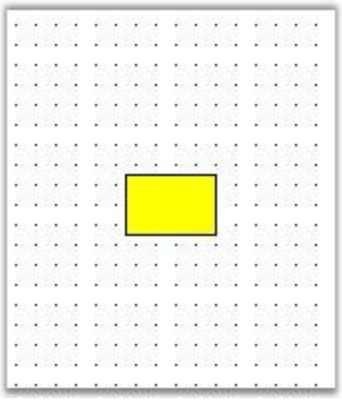

Figure 92: Default Handle Edit mode with Resize

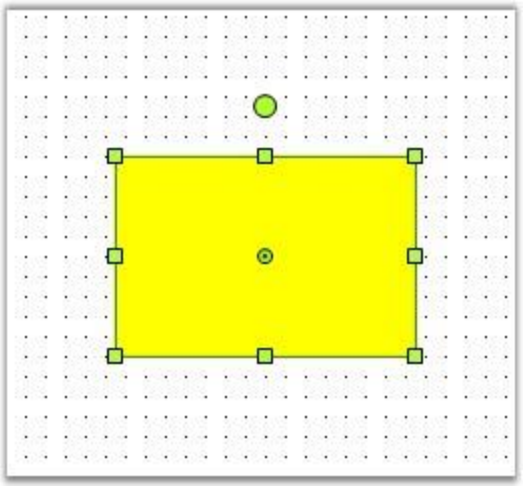

Figure 93: Pinpoint and Rotation Handle

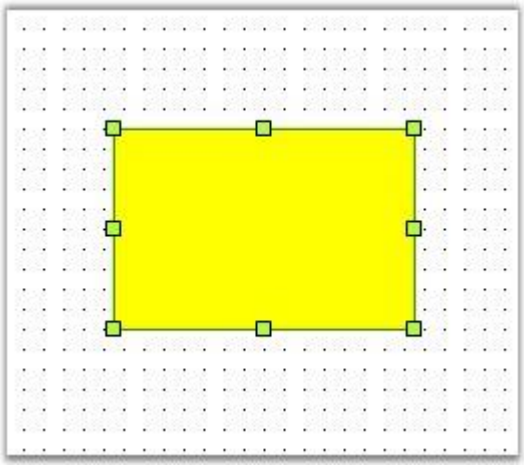

Figure 94: Hide Point and Rotation Handle

4.6.2 Ports And Connections

This section elaborates about the

4.6.2.1 Ports

Port is an object used to establish a connection between the node and the link.

Central Port

By default, the central port for a diagram is enabled using the **EnableCentralPort** property available for the node.

Property	Description
EnableCentralPort	Used to enable or disable the CentralPort.

The central port for a diagram node can be enabled by using the following code snippet.

```
[C#]

Ellipse ellips = new Ellipse(100, 100, 200, 100);
ellips.EnableCentralPort = true;
```

```
[VB]

Dim ellips As New Ellipse(100, 100, 200, 100)
ellips.EnableCentralPort = True
```

In the above code snippets, the Central Port is enabled for an Ellipse node.

Sample diagram is as follows:

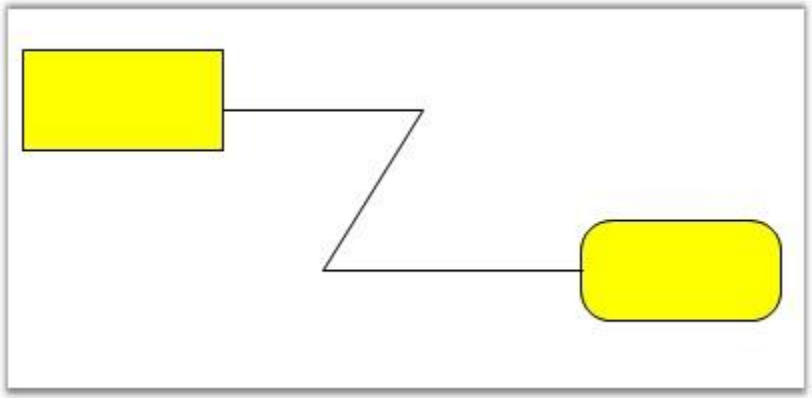

Figure 95: Central Port

Custom ports

Custom ports can be defined at any position of the diagram node, thus allowing the creation of any number of connection ports at any position on the node. All the connections can be defined from the required point or port. Unlike the default port, the custom port when set, will be visible. The **DrawPorts** property must be enabled for custom ports to be created.

Note: When a link is drawn to a node or another link and when the EnableCentralPort is set to True, the links cannot be connected to the custom port. Hence make sure to disable that property for the links and the nodes to connect the links to the custom ports.

Property	Description
DrawPorts	Specifies whether creation of custom ports is enabled. Default value is True.

The Syncfusion.Windows.Forms.Diagram.ConnectionPoint class is used to create custom ports and define their properties. For details, see ConnectionPoint Properties.

The following code snippet illustrate the Custom Ports,


```
[C#]

Syncfusion.Windows.Forms.Diagram.Rectangle rect = new
Syncfusion.Windows.Forms.Diagram.Rectangle(100, 100, 100, 50);
rect.DrawPorts = true;
Syncfusion.Windows.Forms.Diagram.ConnectionPoint cp = new
Syncfusion.Windows.Forms.Diagram.ConnectionPoint();
rect.Ports.Add(cp);
```

```
[VB]

Dim rect As New Syncfusion.Windows.Forms.Diagram.Rectangle(100, 100,
100, 50)
rect.DrawPorts = True
Dim cp As New Syncfusion.Windows.Forms.Diagram.ConnectionPoint()
rect.Ports.Add(cp)
```

Sample diagram is as follows.

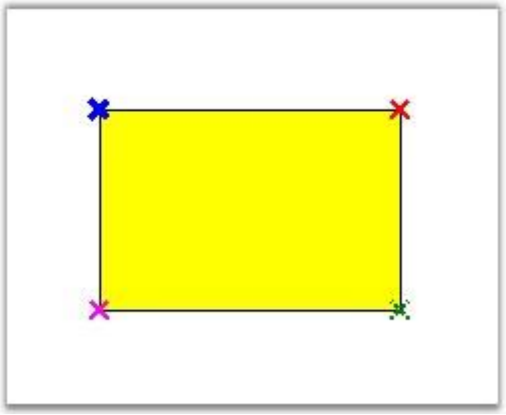

Figure 96: Rectangle Node with Four Custom Ports

Port Shapes

The **VisualType** property available for the port can be used for customizing the shape of the port. There are several types of ports available for customizing the port's shape, each of which differs depending on how they are positioned within the symbol and how they are rendered. For example, a CirclePort can be positioned anywhere within the bounds of a symbol and renders itself as a circle containing cross hairs. Another example is a CenterPort, which always positions itself in the center of the symbol and has no visual representation.

Property	Description

VisualType	The default value is XPort. The options included are as follows: • CirclePort • XPort • TrianglePort • SquarePort • RhombPort • Custom

The visual types for a port can be defined using the following code snippet.

```
[C#]

port.VisualType = PortVisualType.RhombPort;
```

```
[VB]

port.VisualType = PortVisualType.RhombPort
```

Sample diagram is as follows,

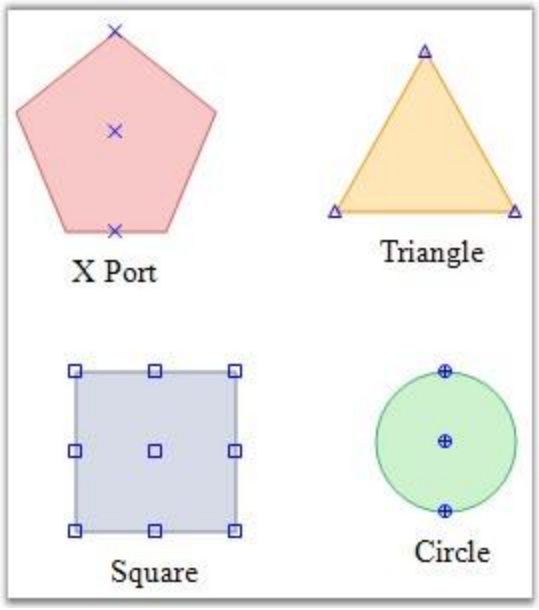

Figure 97: Different Port Shapes

4.6.2.2 Connection Point Properties

ConnectionPoint class provides points to connect to other nodes using a connector. It is available in different custom appearance and in different sizes.

The **ConnectionPointType** and **ConnectionsLimit** properties are available for the ports to define their nature.

Property	Description
ConnectionPointType	Specifies the type of connection to be used. The values included are as follows: • IncomingOutgoing (default) • Outgoing • Incoming
ConnectionsLimit	Specifies the number of connections to be allowed. Default value is 10.

The following code snippet demonstrates their usage.

```
[C#]

Syncfusion.Windows.Forms.Diagram.ConnectionPoint cp = new
Syncfusion.Windows.Forms.Diagram.ConnectionPoint();
cp.ConnectionPointType = ConnectionPointType.Incoming;
cp.ConnectionsLimit = 12;
```

```
[VB]

Dim cp As New Syncfusion.Windows.Forms.Diagram.ConnectionPoint()
cp.ConnectionPointType = ConnectionPointType.Incoming
cp.ConnectionsLimit = 12
```

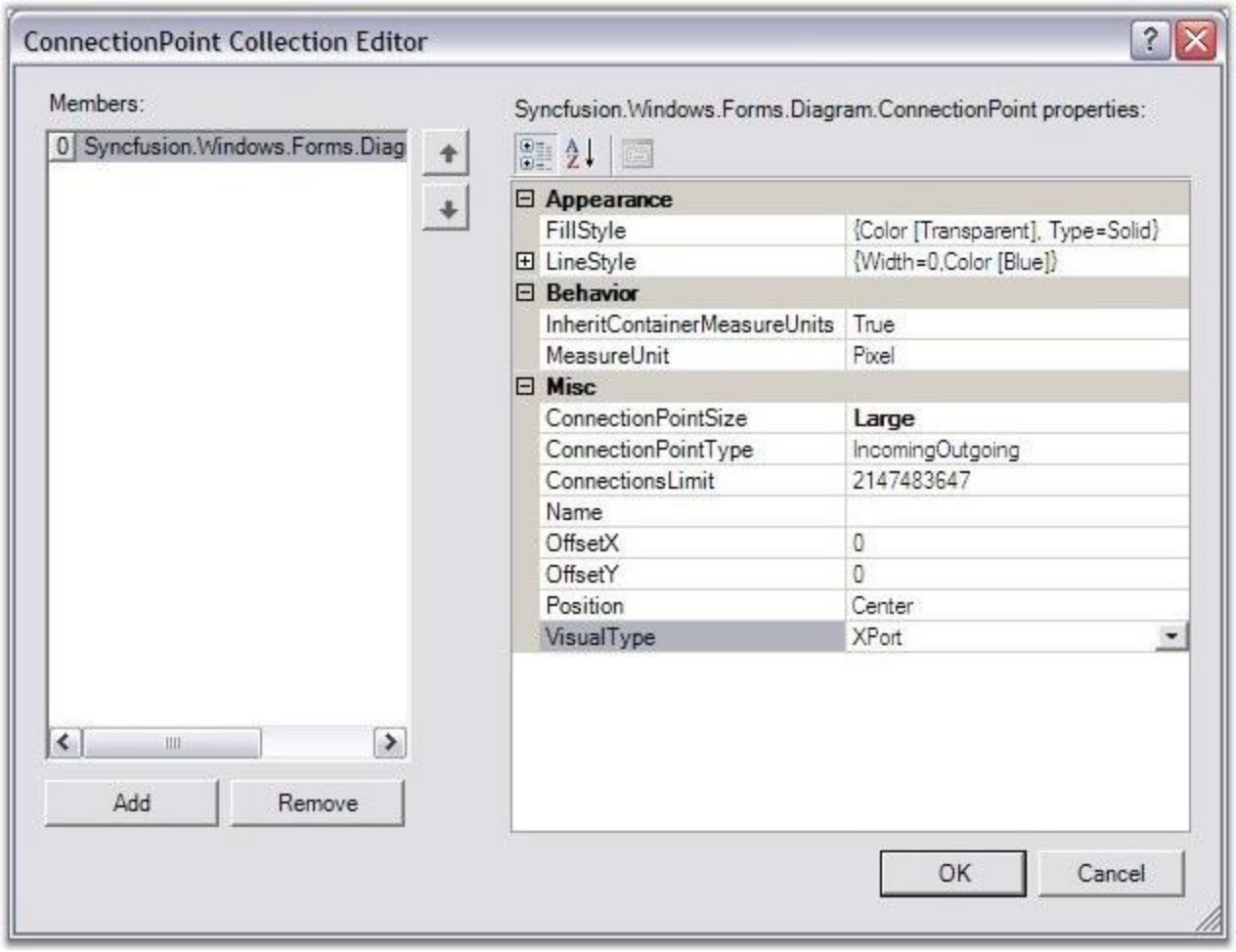

Figure 98: ConnectionPoint Collection Editor

Sample diagram is as follows:

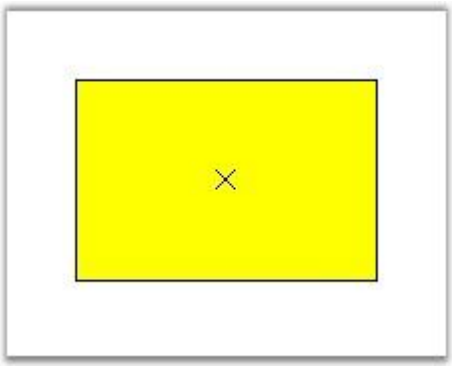

Figure 99: Rectangle with ConnectionPoint

Some important properties are discussed below:

FillStyle

FillStyle property is used to create brushes for filling the interior region of the Connection Points.

```
[C#]

FillStyle m_styleFill = new FillStyle();
m_styleFill.Color = Color.Transparent;
m_styleFill.Type = FillStyleType.Solid;
m styleFill.ColorAlphaFactor = 60;
```

```
[VB]

Dim m_styleFill As New FillStyle()
m_styleFill.Color = Color.Transparent
m_styleFill.Type = FillStyleType.Solid
m_styleFill.ColorAlphaFactor = 60
```

The following image illustrates the above settings.

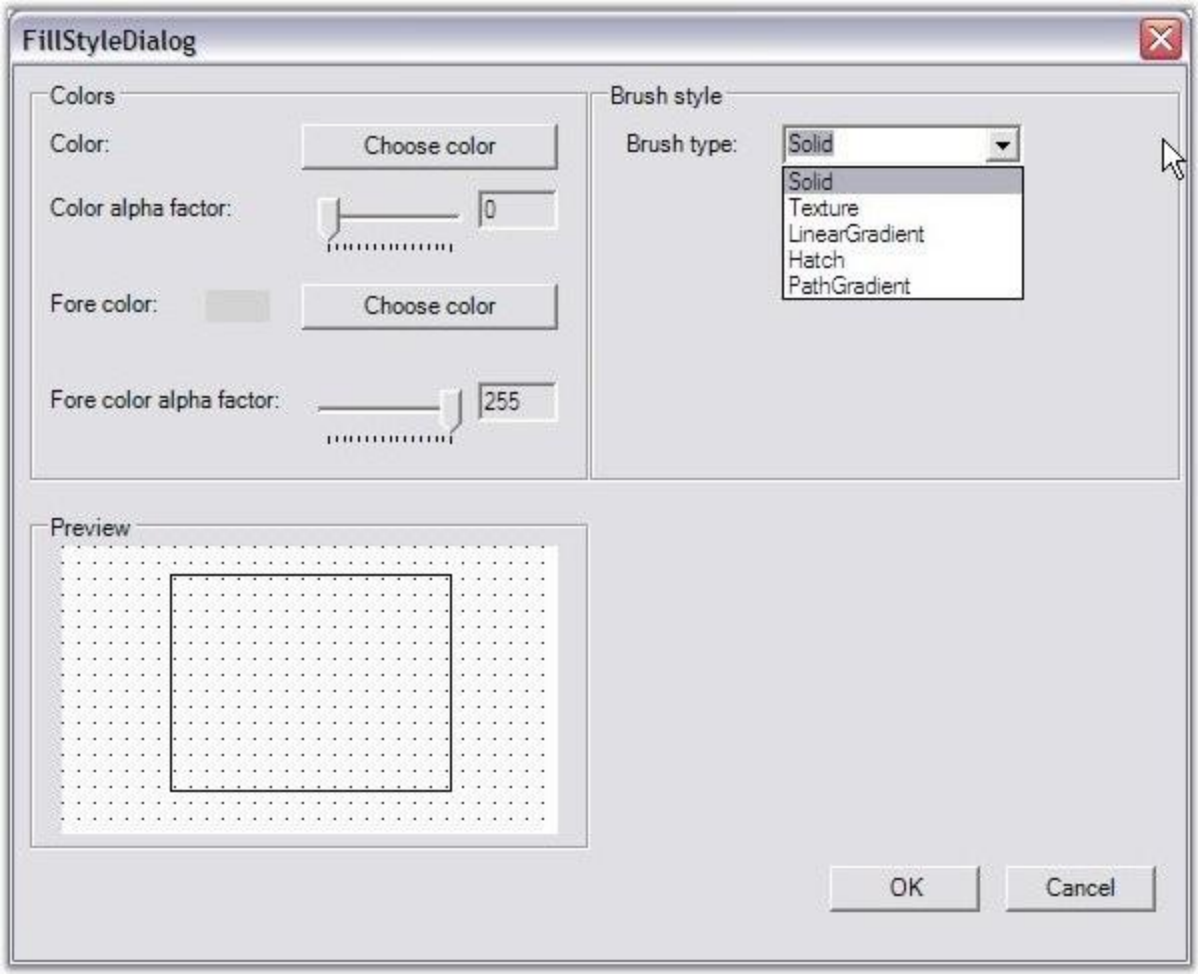

Figure 100: FillStyle Dialog Box

LineStyle

This property inturn has customization properties to set the style for the Connection Point Lines, similar to the other line types.

Figure 101: Line Style

```
[C#]

m_styleLine = new LineStyle();
m_styleLine.LineColor = Color.Blue;
m_styleLine.LineWidth = 0;
m_styleLine.DashStyle = DashStyle.Dash;
```

```
[VB]

m_styleLine = New LineStyle()
m_styleLine.LineColor = Color.Blue
m_styleLine.LineWidth = 0
m_styleLine.DashStyle = DashStyle.Dash
```

The below images illustrates the above settings.

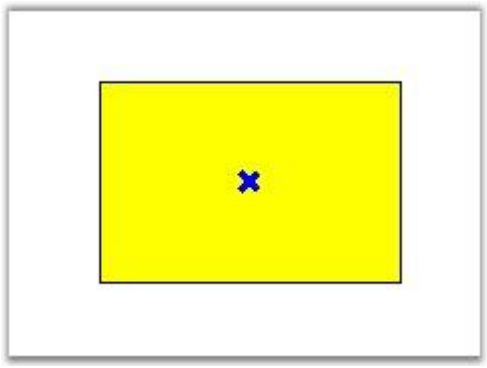

Figure 102: Customized Connection Point

ConnectionPointSize

This property allows us to set the size of the Ports for current ConnectionPoint. This property accepts a ConnectionPointSize enumerator which has three predefined sizes as follows.

Large(12 * 12), Medium (9 *9) & Small (6 * 6).

Position

The point at which the connection should be established can be easily customized by setting the Position property to one of the options. This automatically associates the link to the desired position. Offset values can be specified through OffsetX and OffsetY properties, which will be inherited when the Position is set to Custom.

Properties	Description
OffsetX	Specifies the position which takes the x value of the node. It positions the link with respect to the x value of the node.
OffsetY	Specifies the Y offset value where the link should be aligned. It positions the link with respect to the Y value of the node.
Position	Specifies the position where the links should be connected to the node. Default value is Center. The options included are as follows: • Center • TopLeft • TopCenter • TopRight • MiddleLeft • MiddleRight • BottomLeft • BottomCenter • BottomRight • Custom

The following code snippet defines the setting of the position values for a node's port.

[C#]

```
Syncfusion.Windows.Forms.Diagram.ConnectionPoint cp = new
Syncfusion.Windows.Forms.Diagram.ConnectionPoint();
cp.Position = Position.BottomLeft;
cp.OffsetX = 50;
cp.OffsetY = 10;
```

[VB]

```
Dim cp As New Syncfusion.Windows.Forms.Diagram.ConnectionPoint()
cp.Position = Position.BottomLeft
cp.OffsetX = 50
cp.OffsetY = 10
```

Sample diagram is as follows,

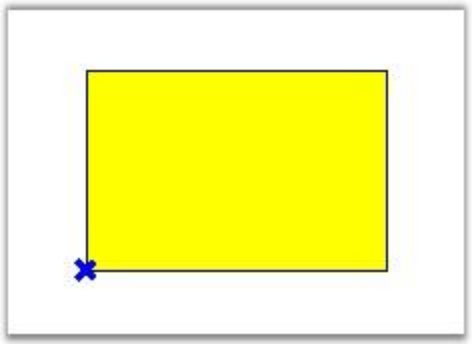

Figure 103: BottomLeft ConnectionPoint

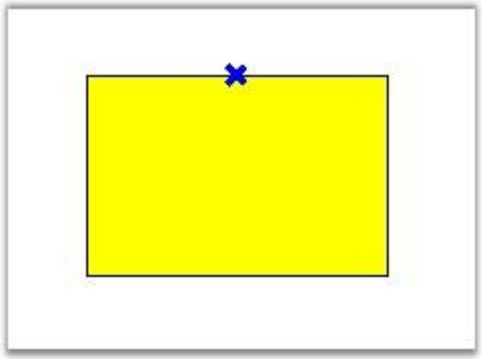

Figure 104: TopCenter ConnectionPoint

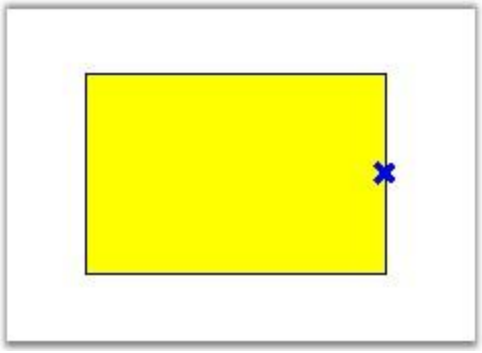

Figure 105: MiddleRight ConnectionPoint

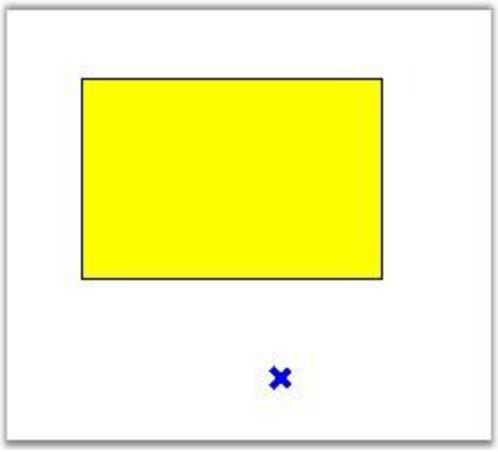

Figure 106: ConnectionPoint in Specified X & Y Offset

4.6.2.2.1 Reject Connections

This feature allows the ConnectionPoint to discard the incoming or outgoing connections to or from that point by setting the ConnectionPointType as Reject. The prohibition sign will be shown when users attempt to connect a line connector to it.

The following code sample illustrates how to reject the incoming and outgoing connections from the ConnectionPoint:

```
[C#]
ConnectionPoint port = new ConnectionPoint();
port.Position = Position.MiddleLeft;
// Sets the ConnectionPointType as Reject, which rejects the incoming and outgoing connections.
port.ConnectionPointType = ConnectionPointType.Reject;
rect1.Ports.Add(port);
```

```
[VB]
Dim port As New ConnectionPoint()
port.Position = Position.MiddleLeft
'Sets the ConnectionPointType as Reject, which rejects the incoming and outgoing connections.
port.ConnectionPointType = ConnectionPointType.Reject
rect1.Ports.Add(port)
```


Figure 107: Rejected Connection

4.6.3 Undo / Redo

The actions can be recorded into the history manager such that the undo and redo operations can be performed. The recording can be controlled and the undo and redo actions can be performed using the following tools.

History Manager Tool	Description
Undo	Undo the previous action.
Redo	Redo the previous action. Redo action can be performed only after an undo action.
StartAtomicAction	Stops recording the actions and hence will not be added to the undo history manager.
EndAtomicAction	Cancels the StartAtomicAction process and turns on the recording of actions in the history manager.

Programmatically, it is implemented as follows:

```
[C#]

this.diagram1.Model.HistoryManager.Undo();
this.diagram1.Model.HistoryManager.Redo();
this.diagram1.Model.HistoryManager.StartAtomicAction("Custom Action");
this.diagram1.Model.HistoryManager.EndAtomicAction();
```

```
[VB]

Me.diagram1.Model.HistoryManager.Undo()
Me.diagram1.Model.HistoryManager.Redo()
Me.diagram1.Model.HistoryManager.StartAtomicAction("Custom Action")
```

```
Me.diagram1.Model.HistoryManager.EndAtomicAction()
```

4.6.4 Layers

Layers are transparent sheets that can be added to the model and the objects are added to it. Layers allow to categorically arrange a set of nodes onto the diagram.

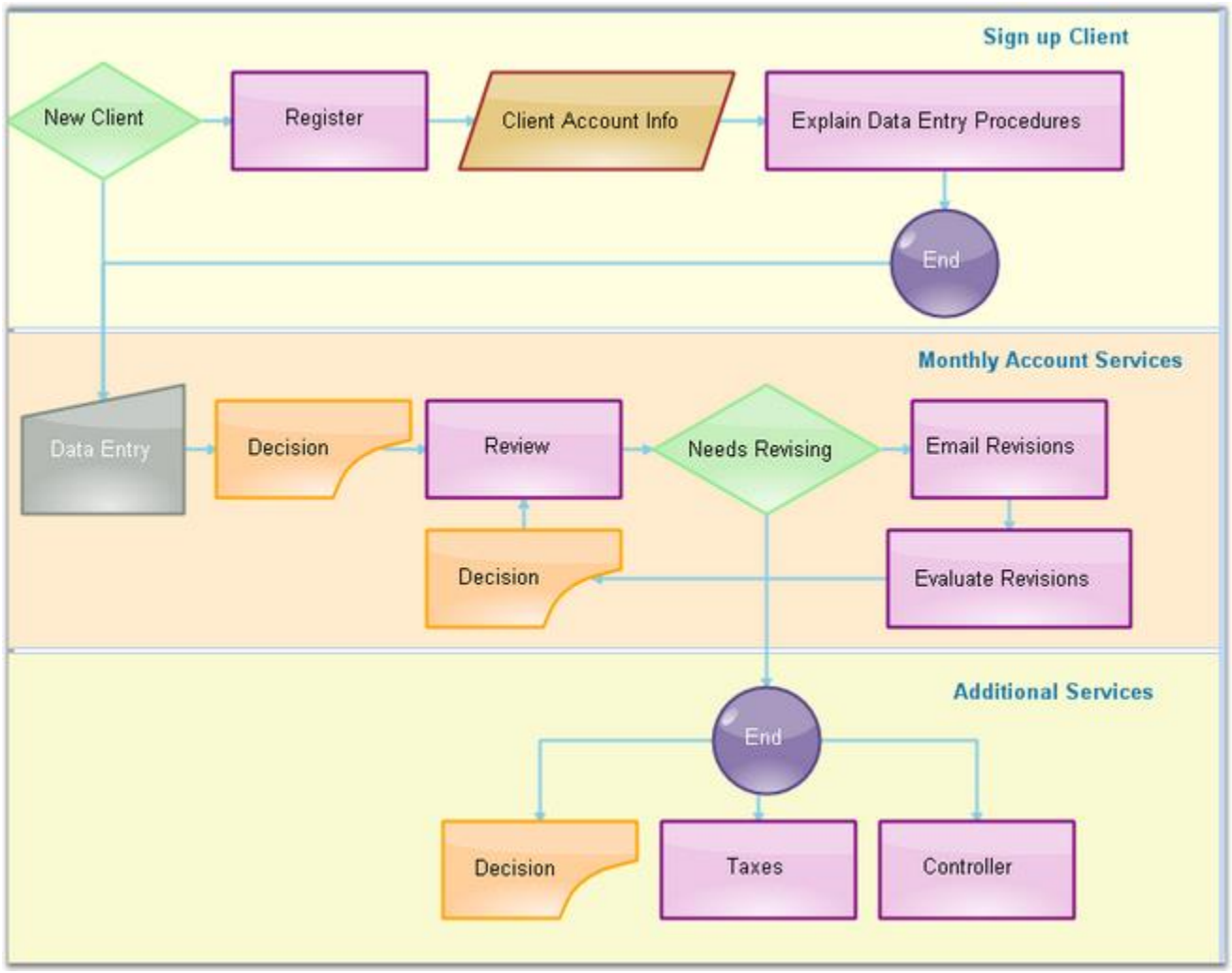

Figure 108: Model Layers

Sample Layers

A layer organizes graphical objects into groups that share a common set of default properties and Z-order. Users can add any number of layers to the model and move objects between layers. Objects in a layer have the same Z-order, which can be relatively controlled with respect to other layers.

Layers are used when the user wants to separate text and links from other nodes. The nodes created can be added to their respective layers. The following code snippet demonstrates the creation of three layers and assigning them to the various nodes.

Layers are used to group nodes in collections and then show or hide layers with all nodes. One node can be assigned to many layers. Node will be visible, if all layers are visible; even if one layer is hidden, node will not be drawn.

Programmatically layers can be implemented as follows.

[C#]

```
// Layer 1

PointF[] pts1 = { new PointF(50, 25), new PointF(75, 75), new
PointF(100, 25), new PointF(125, 75), new PointF(150, 25), new
PointF(175, 75), new PointF(200, 25) };CurveNode cn = new
CurveNode(pts1);
diagram1.Model.AppendChild(cn);
Layer layer1 = new Layer();
cn.Layers.Add(layer1);

// Layer 2

PointF[] pts = { new PointF(10, 100), new PointF(50, 25), new
PointF(34, 78), new PointF(100, 78) };
ClosedCurveNode ccn = new ClosedCurveNode(pts);
diagram1.Model.AppendChild(ccn);
Layer layer2 = new Layer();
ccn.Layers.Add(layer2);

// Layer 3

PointF pt = new PointF(50F, 200F);
PointF pt1 = new PointF(200F, 100F);
BezierCurve bc = new BezierCurve(pt, pt1);
diagram1.Model.AppendChild(bc);
SplineNode sp = new SplineNode(new PointF(130, 200), new PointF(200,
200), new PointF(120, 40));
diagram1.Model.AppendChild(sp);
```

```
Layer layer3 = new Layer();
bc.Layers.Add(layer3);
sp.Layers.Add(layer3);
```

[VB]

```
' Layer 1
Dim pts1 As PointF() = {New PointF(50, 25), New PointF(75, 75), New
PointF(100, 25), New PointF(125, 75), New PointF(150, 25), New
PointF(175, 75), _
 New PointF(200, 25)}
Dim cn As New CurveNode(pts1)
diagram1.Model.AppendChild(cn)
Dim layer1 As New Layer()
cn.Layers.Add(layer1)

' Layer 2
Dim pts As PointF() = {New PointF(10, 100), New PointF(50, 25), New
PointF(34, 78), New PointF(100, 78)}
Dim ccn As New ClosedCurveNode(pts)
diagram1.Model.AppendChild(ccn)
Dim layer2 As New Layer()
ccn.Layers.Add(layer2)

' Layer 3
Dim pt As New PointF(50.0F, 200.0F)
Dim pt1 As New PointF(200.0F, 100.0F)
Dim bc As New BezierCurve(pt, pt1)
diagram1.Model.AppendChild(bc)
Dim sp As New SplineNode(New PointF(130, 200), New PointF(200, 200),
New PointF(120, 40))
diagram1.Model.AppendChild(sp)
Dim layer3 As New Layer()
bc.Layers.Add(layer3)
sp.Layers.Add(layer3)
```


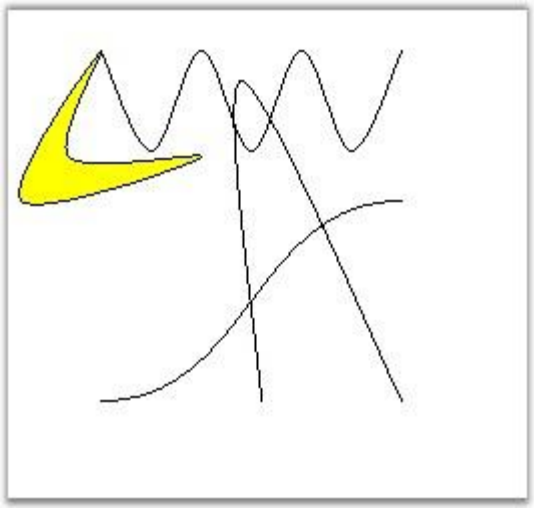

Figure 109: Diagram with Three Layers

Adding Layers

Layers can be added to the model through **LayersCollectionEditor,** which can be opened by selecting the Layers **Collection** property.

Properties	Description
Enabled	Indicates whether the layer should be active or not. Default value is False.
Name	Indicates whether the unit should be inherited.
Visible	Indicates whether the objects on the layer should be visible.

Adding Objects to a Layer

To add objects to a layer, that layer must be active. If an object is added to the model, it will be automatically added to that active layer. The layer can be made active only on setting the **Enabled** property of that layer.

The objects can be added to more than one layer by setting the Enabled property of all the layers to which it is added.

To add an object only to a single layer, make sure that only a single layer is enabled at a time.

Object Visibility

The visibility of the layer can be handled to control the visibility of all the objects on that layer.

```
[C#]

Layer layer0 = new Layer();
this.diagram1.Model.Layers.Add(layer0);
layer0.Enabled = true;
layer0.Visible = true;
layer1.Visible = true;
```

```
[VB]

Dim layer0 As New Layer()
Me.diagram1.Model.Layers.Add(layer0)
layer0.Enabled = True
layer0.Visible = True
layer1.Visible = True
```

4.6.5 Rulers

Rulers can be enabled by setting the **ShowRulers** property for the diagram control. The rulers will automatically inherit the MeasurementUnit set for the diagram model and get converted accordingly.

The height of the ruler can be set through **RulersHeight** property.

Property	Description
ShowRulers	Specifies whether to display ruler for the diagram control.
RulersHeight	Specifies the height of the ruler.

Programmatically the ruler properties can be set as follows.

```
[C#]

this.diagram1.ShowRulers = true;
this.diagram1.RulersHeight = 25;
```

```
[VB]

Me.diagram1.ShowRulers = True
Me.diagram1.RulersHeight = 25
```

Sample diagram is as follows,

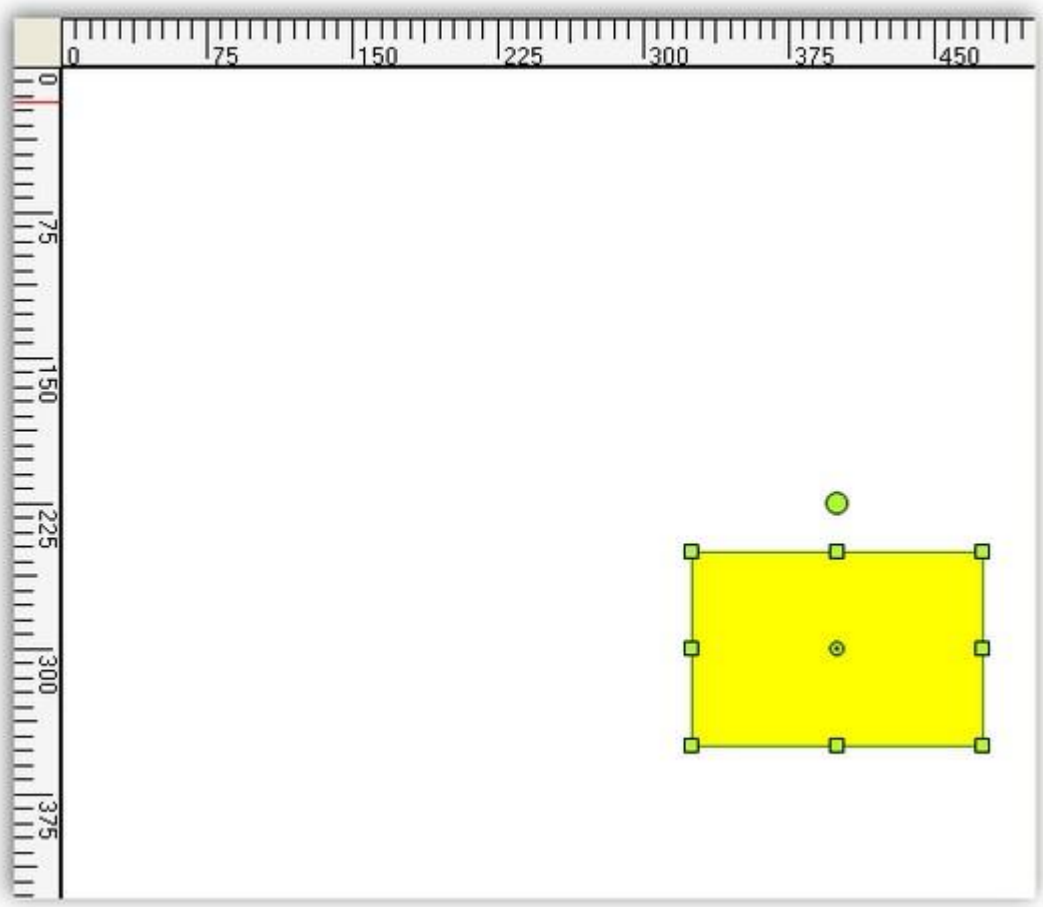

Figure 110: Diagram with Rulers

Diagram With Rulers

Horizontal and vertical rulers can be displayed by setting the **ShowRulers** property. Both the horizontal and vertical rulers can be customized using various properties, which can be separately applied for each of the rulers.

Ruler Property	Description

BackgroundColor	Specifies the back color for the ruler.
HighlightColor	Specifies the highlight color.
MajorLinesColor	Specifies the color for the main line in the ruler.
MarkerColor	Specifies the marker color in the ruler.
MinorLinesColor	Specifies the color for the sub-division lines.
TextStyle	Specifies the text style.

Programmatically the properties can be set as follows for vertical lines.

```
[C#]

this.diagram1.VerticalRuler.BackgroundColor =
System.Drawing.Color.Beige;
this.diagram1.VerticalRuler.HighlightColor =
System.Drawing.Color.Yellow;
this.diagram1.VerticalRuler.MajorLinesColor =
System.Drawing.Color.YellowGreen;
this.diagram1.VerticalRuler.MarkerColor = System.Drawing.Color.Thistle;
this.diagram1.VerticalRuler.MinorLinesColor =
System.Drawing.Color.Turquoise;

this.diagram1.VerticalRuler.TextStyle.Bold = true;
this.diagram1.VerticalRuler.TextStyle.Italic = true;
this.diagram1.VerticalRuler.TextStyle.PointSize = 20;
this.diagram1.VerticalRuler.TextStyle.Strikeout = true;
this.diagram1.VerticalRuler.TextStyle.Style =
System.Drawing.FontStyle.Bold;
this.diagram1.VerticalRuler.TextStyle.Underline = true;
this.diagram1.VerticalRuler.TextStyle.Unit = MeasureUnits.Point;
```

```
[VB]

Me.diagram1.VerticalRuler.BackgroundColor = System.Drawing.Color.Beige
Me.diagram1.VerticalRuler.HighlightColor = System.Drawing.Color.Yellow
Me.diagram1.VerticalRuler.MajorLinesColor =
System.Drawing.Color.YellowGreen
Me.diagram1.VerticalRuler.MarkerColor = System.Drawing.Color.Thistle
Me.diagram1.VerticalRuler.MinorLinesColor =
System.Drawing.Color.Turquoise
```

```
Me.diagram1.VerticalRuler.TextStyle.Bold = True
Me.diagram1.VerticalRuler.TextStyle.Italic = True
Me.diagram1.VerticalRuler.TextStyle.PointSize = 20
Me.diagram1.VerticalRuler.TextStyle.Strikeout = True
Me.diagram1.VerticalRuler.TextStyle.Style =
System.Drawing.FontStyle.Bold
Me.diagram1.VerticalRuler.TextStyle.Underline = True
Me.diagram1.VerticalRuler.TextStyle.Unit = MeasureUnits.Point
```

Sample diagram is as follows,

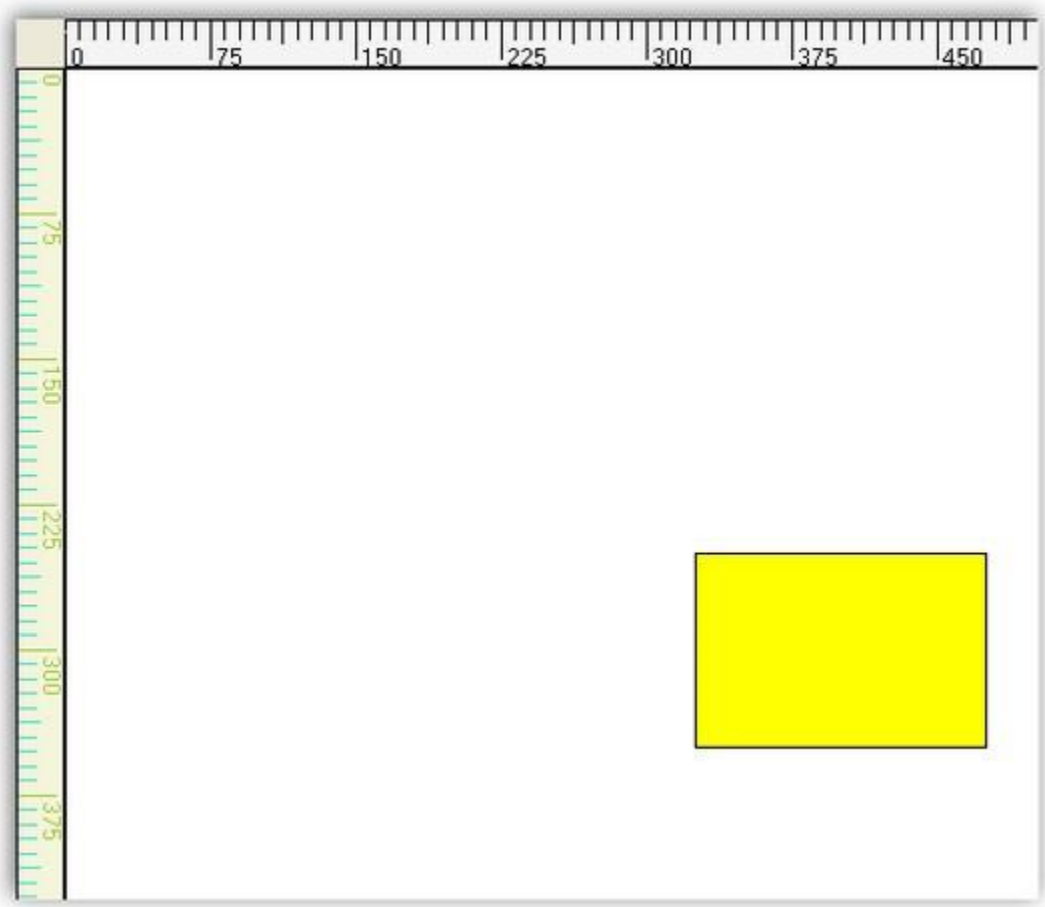

Figure 111: Vertical ruler Property Settings

These properties can be set separately for the horizontal ruler by using HorizontalRuler instead of VerticalRuler as follows.

```
[C#]
```

```
this.diagram1.HorizontalRuler.BackgroundColor =
System.Drawing.Color.Beige;
this.diagram1.HorizontalRuler.HighlightColor =
System.Drawing.Color.Yellow;
this.diagram1.HorizontalRuler.MajorLinesColor =
System.Drawing.Color.YellowGreen;
this.diagram1.HorizontalRuler.MarkerColor =
System.Drawing.Color.Thistle;
this.diagram1.HorizontalRuler.MinorLinesColor =
System.Drawing.Color.Turquoise;

this.diagram1.HorizontalRuler.TextStyle.Bold = true;
this.diagram1.HorizontalRuler.TextStyle.Italic = true;
this.diagram1.HorizontalRuler.TextStyle.PointSize = 20;
this.diagram1.HorizontalRuler.TextStyle.Strikeout = true;
this.diagram1.HorizontalRuler.TextStyle.Style =
System.Drawing.FontStyle.Bold;
this.diagram1.HorizontalRuler.TextStyle.Underline = true;
this.diagram1.HorizontalRuler.TextStyle.Unit = MeasureUnits.Point;
```

```
[VB]

Me.diagram1.HorizontalRuler.BackgroundColor =
System.Drawing.Color.Beige
Me.diagram1.HorizontalRuler.HighlightColor =
System.Drawing.Color.Yellow
Me.diagram1.HorizontalRuler.MajorLinesColor =
System.Drawing.Color.YellowGreen
Me.diagram1.HorizontalRuler.MarkerColor = System.Drawing.Color.Thistle
Me.diagram1.HorizontalRuler.MinorLinesColor =
System.Drawing.Color.Turquoise

Me.diagram1.HorizontalRuler.TextStyle.Bold = True
Me.diagram1.HorizontalRuler.TextStyle.Italic = True
Me.diagram1.HorizontalRuler.TextStyle.PointSize = 20
Me.diagram1.HorizontalRuler.TextStyle.Strikeout = True
Me.diagram1.HorizontalRuler.TextStyle.Style =
System.Drawing.FontStyle.Bold
Me.diagram1.HorizontalRuler.TextStyle.Underline = True
Me.diagram1.HorizontalRuler.TextStyle.Unit = MeasureUnits.Point
```

Sample diagram is as follows,

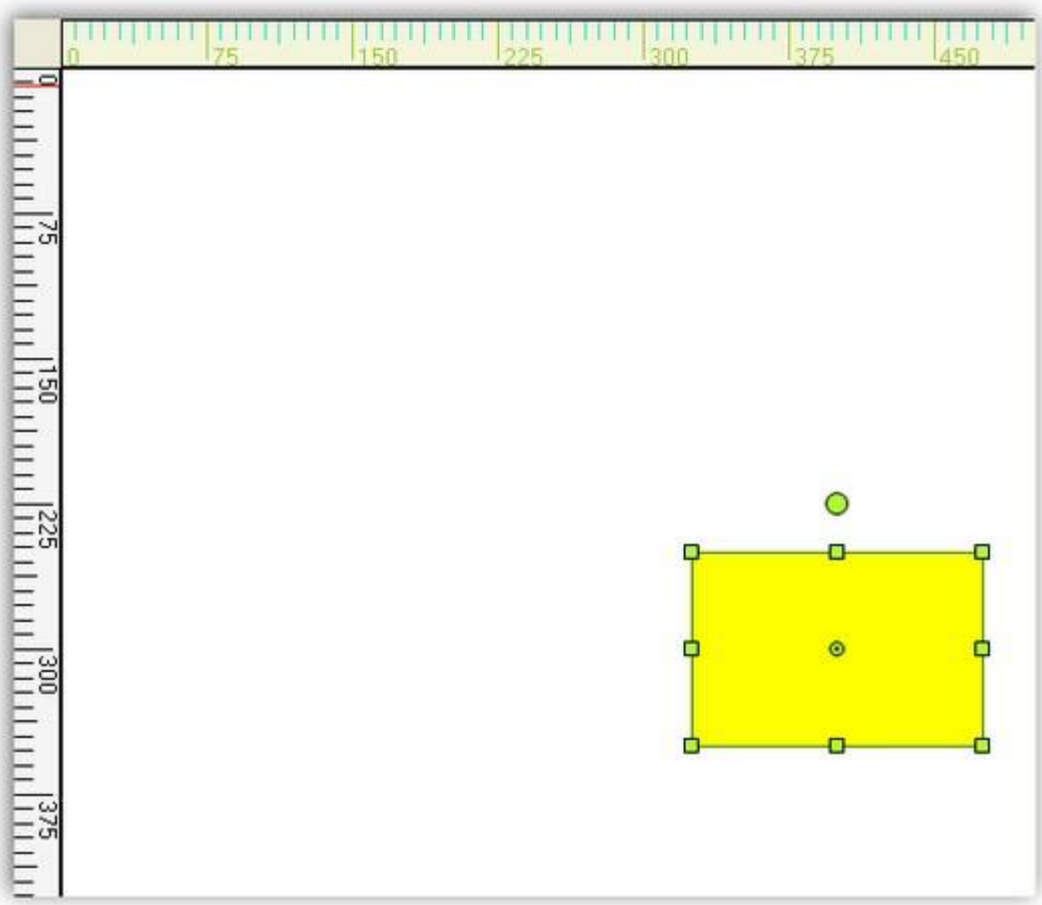

Figure 112: Horizontal Ruler Property Settings

4.6.6 Grouping

A group is a node that acts as a transparent container for other nodes. A group is a composite node that controls a set of child nodes. The bounding rectangle of a group is the union of the bounds of its children. The group renders itself by iterating through its children and rendering them. Child nodes cannot be selected or manipulated individually. Members of the group are added and removed through the ICompositeNode interface.

There are two ways available to add a Group in diagram control:

1. Add the children to the group manually with the help of Group class methods. The below code snippet creates a group with two nodes.

```
[C#]
```



```
//Node 1

Syncfusion.Windows.Forms.Diagram.Rectangle nodeRect = new
Syncfusion.Windows.Forms.Diagram.Rectangle(50, 100, 125, 75);
nodeRect.FillStyle.Color = Color.FromArgb(255, 223, 189);
nodeRect.LineStyle.LineColor = Color.Orange;
Syncfusion.Windows.Forms.Diagram.Label lbl = new
Syncfusion.Windows.Forms.Diagram.Label(nodeRect, "Rectangle");
lbl.FontStyle.Size = 12;
lbl.FontStyle.Bold = true;
nodeRect.Labels.Add(lbl);

//Node 2
Syncfusion.Windows.Forms.Diagram.Rectangle nodeRect1 = new
Syncfusion.Windows.Forms.Diagram.Rectangle(150, 100, 125, 75);
nodeRect1.FillStyle.Color = Color.FromArgb(255, 223, 189);
nodeRect1.LineStyle.LineColor = Color.Orange;
Syncfusion.Windows.Forms.Diagram.Label lbl1 = new
Syncfusion.Windows.Forms.Diagram.Label(nodeRect1, "Rectangle1");
lbl1.FontStyle.Size = 12;
lbl1.FontStyle.Bold = true;
nodeRect1.Labels.Add(lbl1);

//Grouping Nodes
Syncfusion.Windows.Forms.Diagram.Group grp = new Group();
grp.AppendChild(nodeRect);
grp.AppendChild(nodeRect1);
this.DiagramWebControl1.Model.AppendChild(grp);
```

2. Diagram control support two direct methods for Grouping and UnGrouping as follows.

```
[C#]

this.diagram1.Controller.Group();     //Method to Group the nodes
this.diagram1.Controller.UnGroup();   //Method to UnGroup the nodes
```

How to access the child nodes in a Group, and how to delete / remove the node

The first step is to check whether the node is a Group.

```
[C#]

if (node is Group)
{
    // Your code here
}
```

If the node is a Group, then following are some special methods.

```
[C#]

public Node GetChild(int childIndex);
public Node GetChildByName(string childName);
public void RemoveAllChildren();
public bool RemoveChild(int childIndex);
public bool RemoveChild(Node nodeToRemove);
public void InsertChild(Node child, int childIndex);
```

Also, Group has an int **ChildCount** property, which returns the child count in a Group. To delete the first element in a Group, use the below code.

```
[C#]

foreach (Node node in Diagram1.Model.Nodes)
{
    if (node is Group) // Check for Group
    {
        Group groupNode = (Group)node;
        if (groupNode.ChildCount > 0) // Group has sub nodes
        {
            Node nodeToRemove = groupNode.GetChild(0);
            groupNode.RemoveChild(nodeToRemove);

        }
    }
}
```

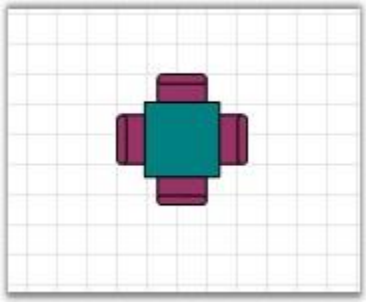

Figure 113: Group Node

4.6.6.1 Positioning nodes in Group

Positioning support

Diagram Group node supports absolute and relative positioning.

Positioning Group node's Child

Group Node has an enum property called **GroupNodePosition** of type **GroupNodePositions** to position its child nodes. GroupNodePositions has two values Absolute and Relative. The Absolute will place the nodes inside a group based on their actual pinpoint whereas the Relative will place the nodes based on their default pinpoint. Default value is *Relative*.

```
[C#]
//Group
Group group = new Group();
//Absolute positioning
group.GroupNodePosition = GroupNodePositions.Absolute;
```

```
[VB]
'Group
Dim group As Group = New Group ()
'Absolute positioning
group.GroupNodePosition = GroupNodePositions.Absolute
```

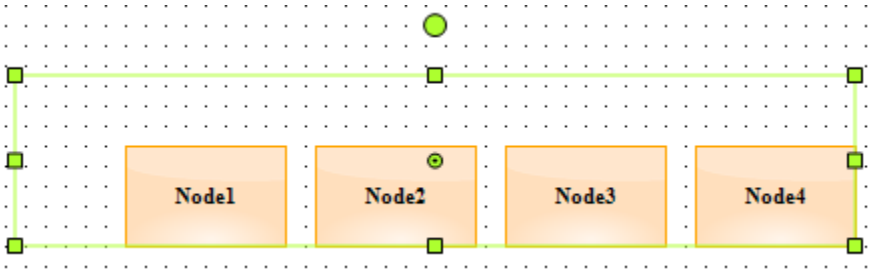

Figure 114: Group Node Absolute positioning

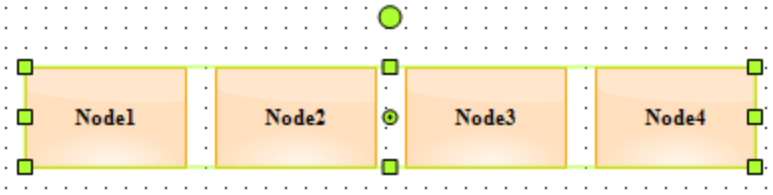

Figure 115: Group Node Relative positioning

Properties

Name	Description	Type	Default value	Value Accepted	Reference
GroupNodePosition	Specifies the mode in which the group node's child should be positioned.	GroupNodePositions	Relative	Absolute, Relative	GroupNodePosition

4.6.7 Scrolling, Zooming And Panning Support

The interactive features like scrolling, zooming and panning support are discussed in this section:

4.6.7.1 Scroll Support

The horizontal and vertical scrollbars can be displayed or hidden by handling the **HScroll** and **VScroll** properties.

Properties	Description
HScroll	Specifies whether to display the horizontal scroll bar.
VScroll	Specifies whether to display the vertical scroll bar.

Programmatically, these properties can be set as follows.

```
[C#]

this.diagram1.HScroll = true;
this.diagram1.VScroll = true;
```

```
[VB]

Me.diagram1.HScroll = True
Me.diagram1.VScroll = True
```

Sample diagram is as follows,

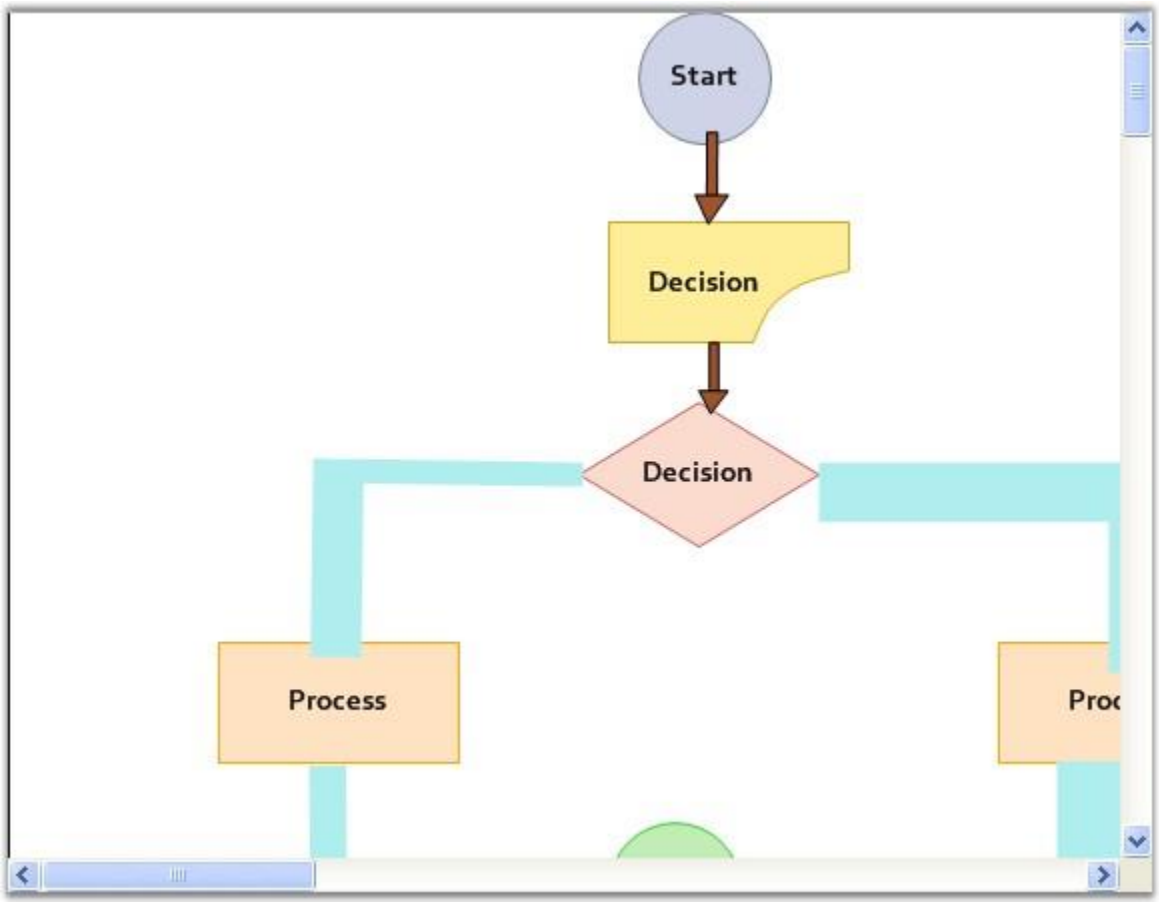

Figure 116: Diagram With Scroll Settings

ScrollGranularity determines the level of granularity for scrolling. The value of this property must be greater than 0. This value is multiplied by virtual size of the view in order to get the scroll range. For example, if the virtual size of the view is 100x50 and this property is set to 0.5f, then the horizontal scroll range is set to 0.50 and the vertical scroll range is set to 0.25.

SmoothMouseWheelScrolling specifies whether the control should perform one scroll command (faster) or if it should perform multiple scroll commands with smaller increments (smoother) when user rolls mouse wheel.

Property	Description
EnableIntelliMouse	Toggles support for Intelli-Mouse panning. When the user presses the middle-mouse button and drags the mouse, the window will scroll.

ScrollGranularity	Specifies the level of granularity for scrolling. This value is to scale the scroll range of the scrollbars.
SmoothMouseWheelScrolling	Specifies whether the control should perform one scroll command (faster) or if it should perform multiple scroll commands with smaller increments (smoother) when user rolls mouse wheel.
HScrollBar	Returns a reference to an object with horizontal scrollbar settings of the control.

Programmatically these properties can be set as follows.

```
[C#]

this.diagram1.EnableIntelliMouse = true;
this.diagram1.ScrollGranularity = .9F;
this.diagram1.SmoothMouseWheelScrolling = false;
this.diagram1.HScrollBar.SmallChange = 200;
```

```
[VB]

Me.diagram1.EnableIntelliMouse = True
Me.diagram1.ScrollGranularity = .9F
Me.diagram1.SmoothMouseWheelScrolling = False
Me.diagram1.HScrollBar.SmallChange = 200
```

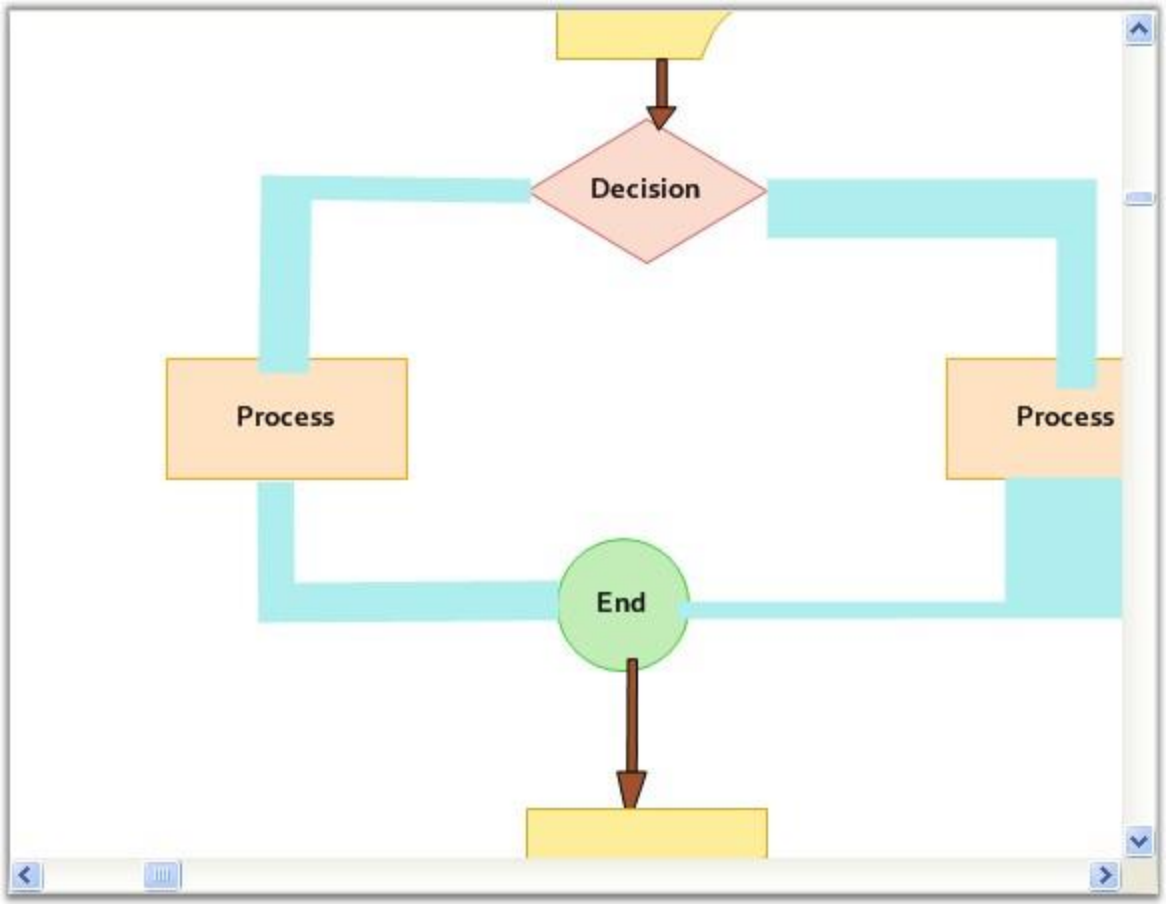

Figure 117: Scroll Properties

Scrollable Area

Diagram has **ScrollVirtualBounds** property, which determines the bounds of the scrollable area. This sets the Diagram control's virtual space i.e, gray area around the control. Sometimes, we may need to remove that area and use Diagram control area alone.

```
[C#]

this.diagram1.ScrollVirtualBounds = new RectangleF(0, 0, 0, 0);
```

```
[VB]

Me.diagram1.ScrollVirtualBounds = New RectangleF(0, 0, 0, 0)
```

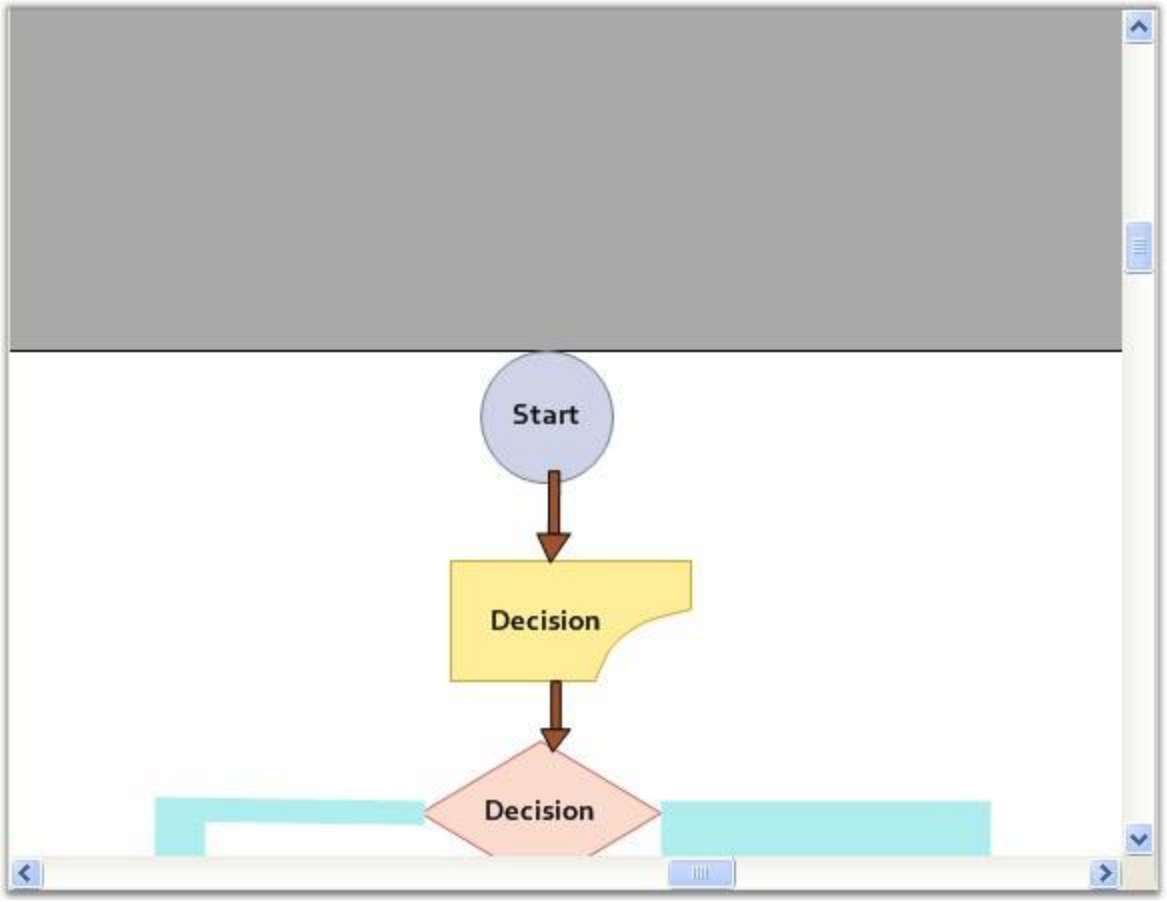

Figure 118: Scroll Virtual Bounds

Scroll Behavior

Scrolling behavior can be controlled by setting the **AccelerateScrolling** property.

Property	Description
AccelerateScrolling	Specifies the scrolling behavior. The options included are as follows: None Default Fast Immediate

AllowIncreaseSmallChange	Specifies if the scroll control can increase the **ScrollBar.SmallChange** property during accelerated scrolling.

Here, setting the AccelerateScrolling to Fast, will increase the scroll speed when the horizontal or vertical thumb is pressed continuously.

```
[C#]

this.diagram1.AccelerateScrolling =
Syncfusion.Windows.Forms.AccelerateScrollingBehavior.Fast;
this.diagram1.AllowIncreaseSmallChange = true;
```

```
[VB]

Me.diagram1.AccelerateScrolling =
Syncfusion.Windows.Forms.AccelerateScrollingBehavior.Fast
Me.diagram1.AllowIncreaseSmallChange = True
```

ThumbTrack

The **HorizontalThumbTrack** and **VerticalThumbTrack** properties allows to handle whether the scroll bar thumb should be used for scrolling.

Property	Description
HorizontalThumbTrack	Specifies if the control should scroll while the user is dragging a horizontal scrollbar thumb.
VerticalThumbTrack	Specifies if the control should scroll while the user is dragging a vertical scrollbar thumb.

Programmatically, these properties can be set as follows.

```
[C#]

this.diagram1.HorizontalThumbTrack = true;
this.diagram1.VerticalThumbTrack = true;
```

```
[VB]

Me.diagram1.HorizontalThumbTrack = True
Me.diagram1.VerticalThumbTrack = True
```

ScrollTips

ScrollTips can be enabled or disabled for horizontal and vertical scroll bars individually by setting the **HorizontalScrollTips** and **VerticalScrollTips** properties.

The format in which the scrolltip should be displayed can be specified using the **ScrollTipFormat** property. The default format is 'Position{0}'.

Properties	Description
HorizontalScrollTips	Specifies whether to display the horizontal scroll bar.
VerticalScrollTips	Specifies whether to display the vertical scroll bar.
ScrollTipFormat	Specifies the format for the scrolltip to be displayed.

Programmatically these properties can be set as follows.

```
[C#]

this.diagram1.HorizontalScrollTips = true;
this.diagram1.VerticalScrollTips = true;
this.diagram1.ScrollTipFormat = "Offset{0}";
```

```
[VB]

Me.diagram1.HorizontalScrollTips = True
Me.diagram1.VerticalScrollTips = True
Me.diagram1.ScrollTipFormat = "Offset{0}"
```

Using Splitter control

When splitter control is used and one or more diagram controls are added, setting the FillSplitterPane docks the diagram control inside the splitter control and fills the entire space.

Property	Description
FillSplitterPane	Specifies whether to fill the splitter control with diagram.

Programmatically these properties can be set as follows.

```
[C#]

this.diagram1.FillSplitterPane = true;
```

```
[VB]

Me.diagram1.FillSplitterPane = True
```

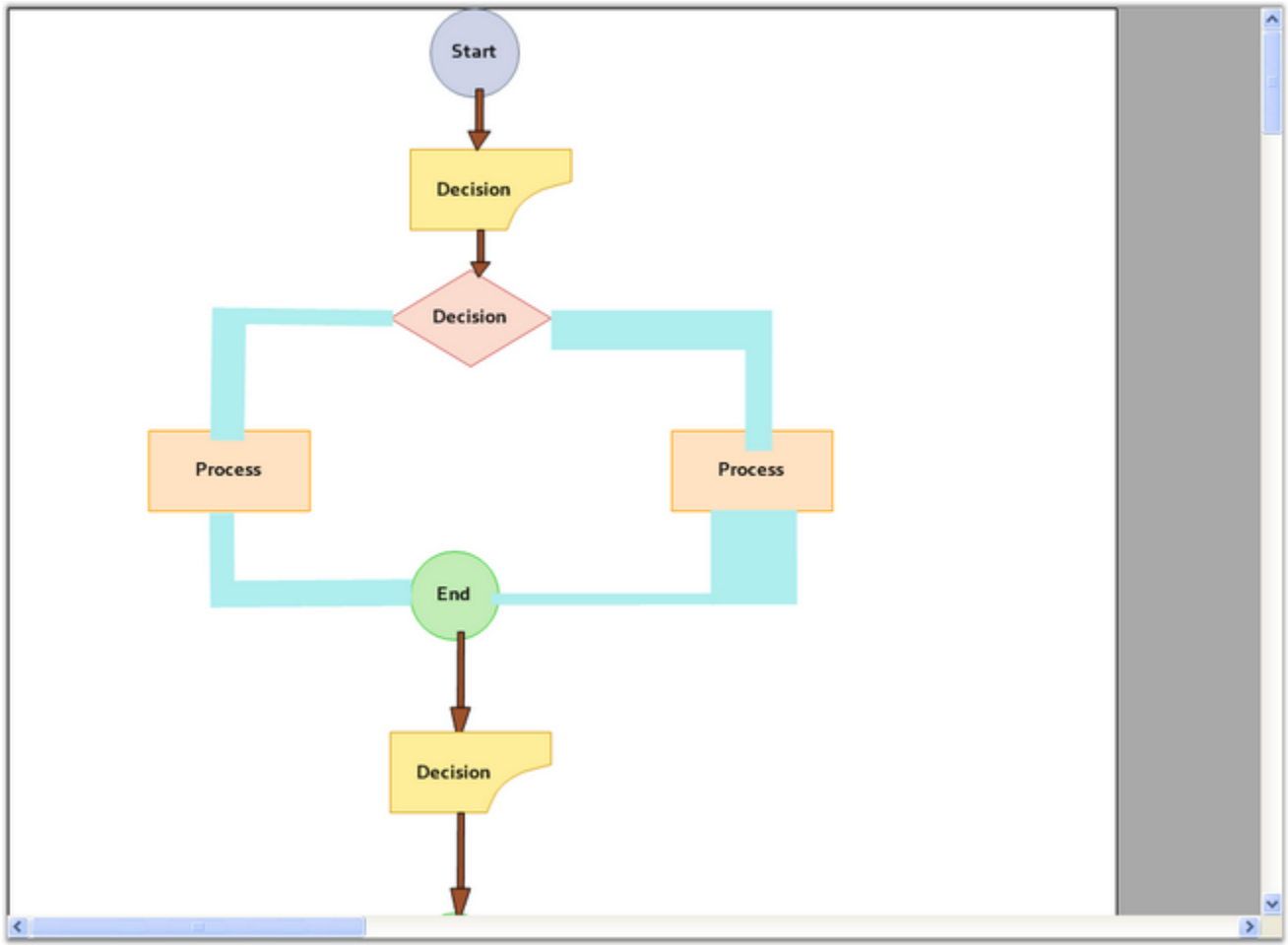

Figure 119: Scroll With Splitter Control

4.6.7.2 Zoom Support

One of the interactive features of Essential Diagram is its zooming capabilities. This feature allows you to interactively zoom in and out of the diagram in the following ways:

- Zoom to the center of the diagram.
- Zoom to the top left of the diagram.
- Zoom to the pointer position using Ctrl and the mouse wheel.

Use Case Scenarios

Users can zoom in and out of diagram content based on their requirements.

Properties

Property	Description	Data Type
ZoomType	Gets or sets the type of zooming to be performed.	enum
ZoomIncrement	Specifies the amount to zoom each time the diagram is zoomed in or out.	float

Methods

Method	Description	Parameters	Return Type
ZoomIn	Zoom in on the diagram document.	NA	void
ZoomOut	Zoom out of the diagram document.	NA	void
ZoomToSelection	Zoom the diagram document to the specified selection bounds.	RectangleF	void
ZoomToActual	Zoom the document to its actual size.	NA	void

4.6.7.2.1 ZoomIn, ZoomOut, ZoomToActual, ZoomToSelection

The diagram document can be zoomed in, zoomed out, zoomed to its original size, and zoomed to a selected area based on the **ZoomIncrement**. You can use the following methods to zoom in the diagram document.

- ZoomIn()
- ZoomOut()
- ZoomToSelection(RectangleF)

- ZoomToActual()

The following code samples explain how to use the zoom methods to zoom in the diagram programmatically:

```
[C#]
//Sets the zoom increment value.
this.diagram1.View.ZoomIncrement = 20;
// Zoom in on the document.
this.diagram1.View.ZoomIn();
// Zoom out of the document.
this.diagram1.View.ZoomOut();
// Zoom the document to its actual size.
this.diagram1.View.ZoomToActual();
// Zoom the document to the selection.
this.diagram1.View.ZoomToSelection(new RectangleF(100,100,100,100));
```

```
[VB]
'Sets the zoom increment value.
Me.diagram1.View.ZoomIncrement = 20
'Zoom in on the document.
Me.diagram1.View.ZoomIn()
'Zoom out of the document.
Me.diagram1.View.ZoomOut()
'Zoom the document to its actual size.
Me.diagram1.View.ZoomToActual()
'Zoom the document to the selection.
Me.diagram1.View.ZoomToSelection(New RectangleF(100,100,100,100))
```

4.6.7.2.2 Zooming to the Center of the Diagram

The diagram document can be zoomed to the center of the current viewport by setting the **ZoomType** as **Center**. The default value of **ZoomType** is **Center**.

The following code sample demonstrates how to use the zoom to center feature in a diagram:


```
[C#]
// Sets the ZoomType as 'center'.
this.diagram1.View.ZoomType = ZoomType.Center;
```

```
[VB]
'Sets the ZoomType as 'center'.
Me.diagram1.View.ZoomType = ZoomType.Center
```

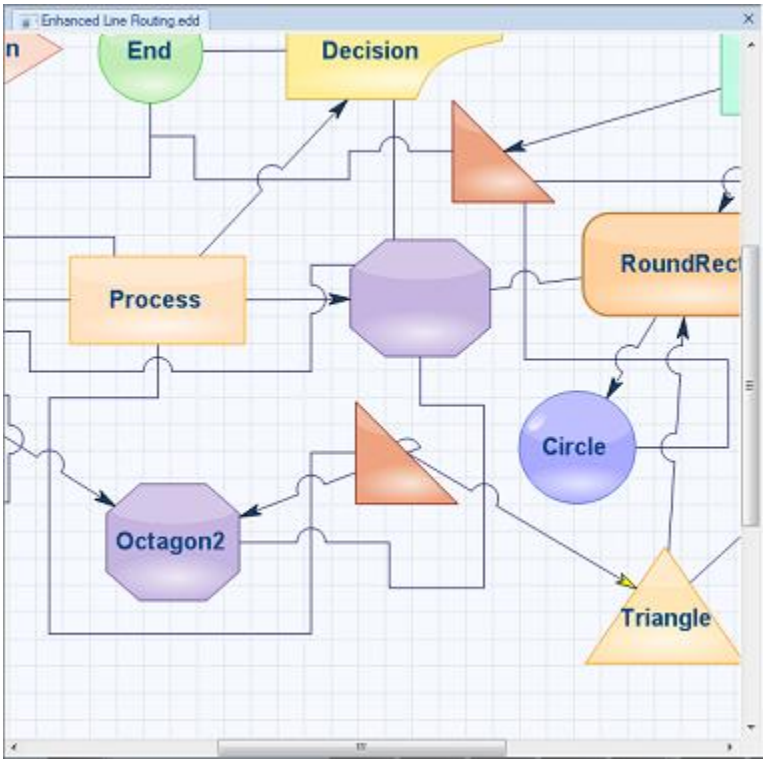

Figure 120: Center-Based Zoom

4.6.7.2.3 Zooming to the Top-Left of the Diagram

The diagram document can be zoomed to the top-left corner of the viewport by setting the **ZoomType** as **TopLeft**.

The following code shows how to use the zoom to top-left feature:

```
[C#]
// Sets the ZoomType as TopLeft.
```



```
this.diagram1.View.ZoomType = ZoomType.TopLeft;
```

```
[VB]
' Sets the ZoomType as TopLeft.
Me.diagram1.View.ZoomType = ZoomType.TopLeft
```

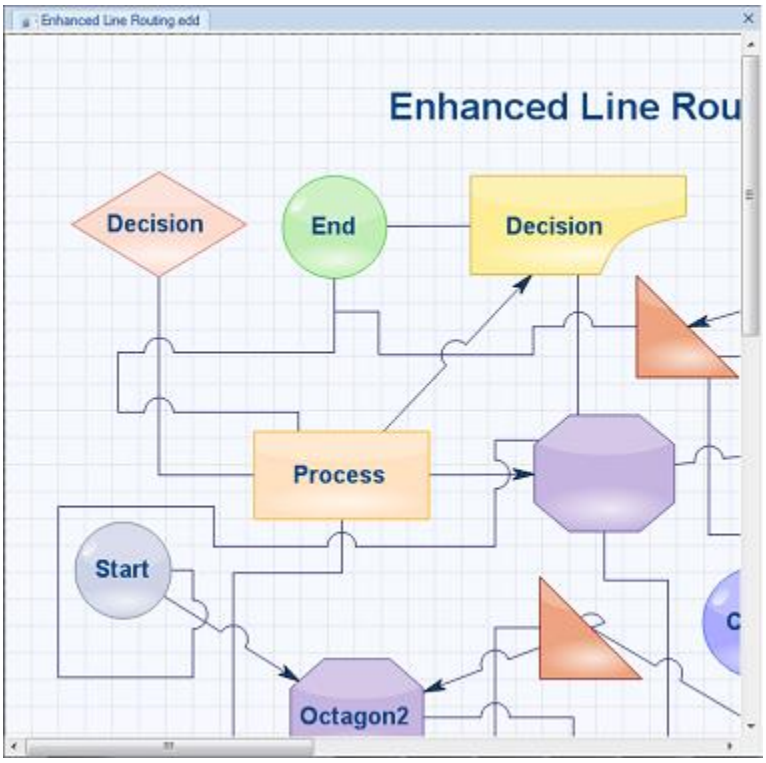

Figure 121: Top-Left Zoom

4.6.7.2.4 Zooming to the Pointer Position

Essential Diagram supports zooming the diagram document to the pointer position using Ctrl and the mouse wheel.

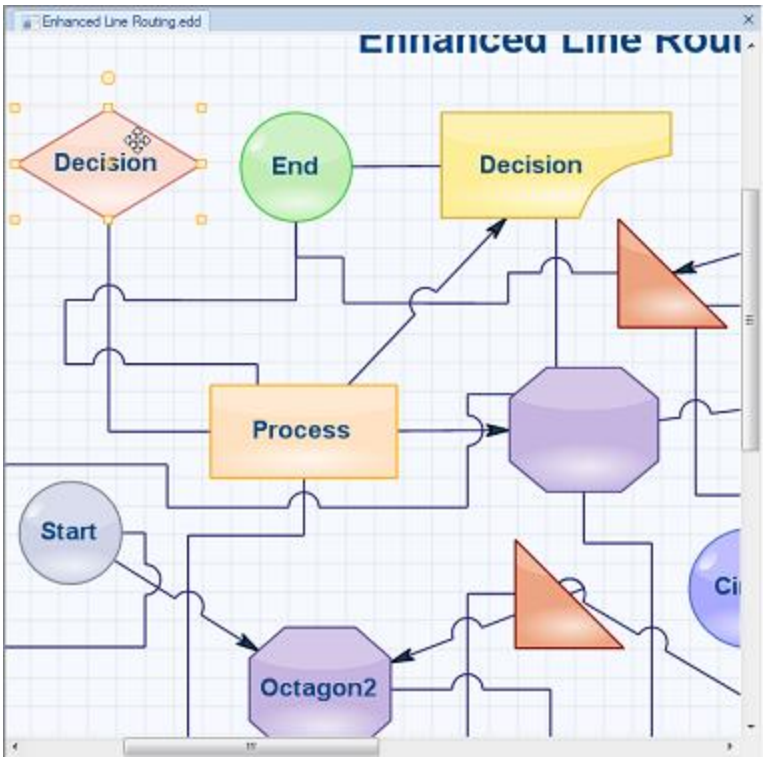

Figure 122: Mouse-based Zoom

4.6.7.2.5 ZoomTool

Essential Diagram supports a UI tool called **ZoomTool** which is used to zoom and select the diagram document interactively. Users can use the ZoomTool's **MaximumMagnification** and **MinimumMagnification** properties to restrict the document's maximum or minimum zoom levels and use the **ZoomIncrement** property to specify the amount to zoom each time the diagram is zoomed in or out.

Zoom Tool Properties

Property	Description	Data Type
MaximumMagnification	Specifies the maximum magnification value for zooming. Default value is 1000.	float
MinimumMagnification	Specifies the minimum magnification value for zooming. Default value is 10.	float
ZoomIncrement	Specifies the amount to zoom each time the mouse is clicked.	float

The following code demonstrates how to activate the zoom tool:


```
[C#]

diagram1.Controller.ActivateTool("ZoomTool");
ZoomTool zoomTool = (ZoomTool)diagram1.Controller.ActiveTool;
zoomTool.MaximumMagnification = 100;
zoomTool.MinimumMagnification = 50;
zoomTool.ZoomIncrement = 10;
```

```
[VB]

diagram1.Controller.ActivateTool("ZoomTool")
Dim zoomTool As ZoomTool = CType(diagram1.Controller.ActiveTool,
ZoomTool)
zoomTool.MaximumMagnification = 100
zoomTool.MinimumMagnification = 50
zoomTool.ZoomIncrement = 10
```


Figure 123: ZoomTool Selection

Figure 124: Zoomed in on the Selected Area

4.6.7.3 Pan Support

Pan tool allows the user to drag the diagram and hence scroll it in any direction.

Programmatically, it is implemented as follows.

```
[C#]

this.diagram1.Controller.ActivateTool("PanTool");
```

```
[VB]

Me.diagram1.Controller.ActivateTool("PanTool")
```

Sample diagram is as follows.

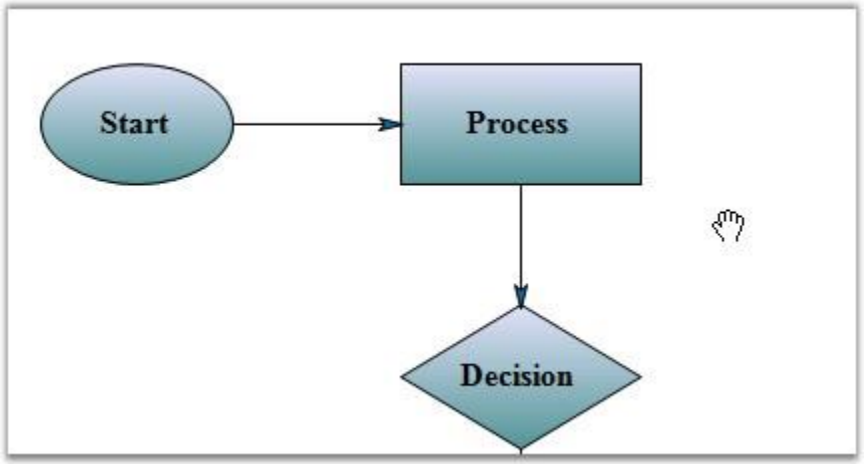

Figure 125: Diagram With Pan Tool

4.6.8 Event Handlers

This section elaborates on the following:

4.6.8.1 Diagram Events

DiagramViewerEventSink class contains the events specific to the diagram. The various events that can be invoked using this class are discussed in the following topics.

4.6.8.1.1 Node Collection Events

This topic discusses the events that are fired while adding or removing the node to or from the node collection. The below table discusses all the available node collection events.

DiagramViewerEventSink	Description
NodeCollectionChanged	Triggered after the node collection changes are completed.
NodeCollectionChanging	Triggered when the node collection is edited.

EventArgs members can be accessed using the following members.

NodeCollection EventArgs Member	Description
Cancel	Indicates whether the NodeCollectionChanged event should be canceled.
ChangeType	It returns the following possible values: • Insert - whether the node is inserted • Remove – whether the node is removed
Element	Returns whether the head or tail end is moved.
Elements	Returns the elements collection on which the event occurs.
Index	Returns the zero-based index into the collection on which the event occurred.
Owner	Returns the base class onto which the node is added.

Inside the NodeCollectionChanged event handler, user can identify whether a node is added or removed from the node collection using simple message box as follows.

```
[C#]

private void Form1_Load(object sender, EventArgs e)
{

    ((DiagramViewerEventSink)diagram1.EventSink).NodeCollectionChanged
+= new
CollectionExEventHandler(Form1_NodeCollectionChanged)((DiagramViewerEve
ntSink)diagram1.EventSink).NodeCollectionChanging += new
```

```
CollectionExEventHandler(Form1_NodeCollectionChanging);


    RectangleF rect = new RectangleF(100, 100, 100, 100);
    RichTextNode richText = new RichTextNode("", rect);
    richText.Text = "Rich text box";

    NodeCollection nodeStack = new NodeCollection();
    nodeStack.Add(richText);
    MessageBox.Show(nodeStack.Count.ToString());

}
private void Form1_NodeCollectionChanging(CollectionExEventArgs e)
{
    MessageBox.Show("NodeCollectionChanging event fired");
}
```

[VB]

```
Private Sub Form1_Load(ByVal sender As Object, ByVal e As EventArgs)

    DirectCast(diagram1.EventSink,
DiagramViewerEventSink).NodeCollectionChanged += New
CollectionExEventHandler(Form1_NodeCollectionChanged)(DirectCast(diagra
m1.EventSink, DiagramViewerEventSink)).NodeCollectionChanging += New
CollectionExEventHandler(Form1_NodeCollectionChanging)


    Dim rect As New RectangleF(100, 100, 100, 100)
    Dim richText As New RichTextNode("", rect)
    richText.Text = "Rich text box"

    Dim nodeStack As New NodeCollection()
    nodeStack.Add(richText)

    MessageBox.Show(nodeStack.Count.ToString())
End Sub
Private Sub Form1_NodeCollectionChanging(ByVal e As
CollectionExEventArgs)
    MessageBox.Show("NodeCollectionChanging event fired")
End Sub
```

Sample diagrams are as follows,

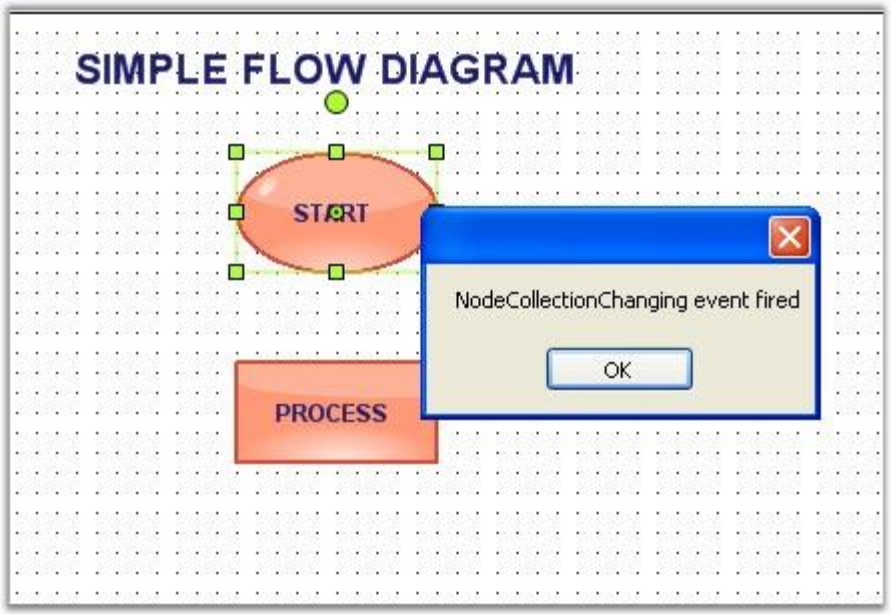

Figure 126: Node Collection Changing Event

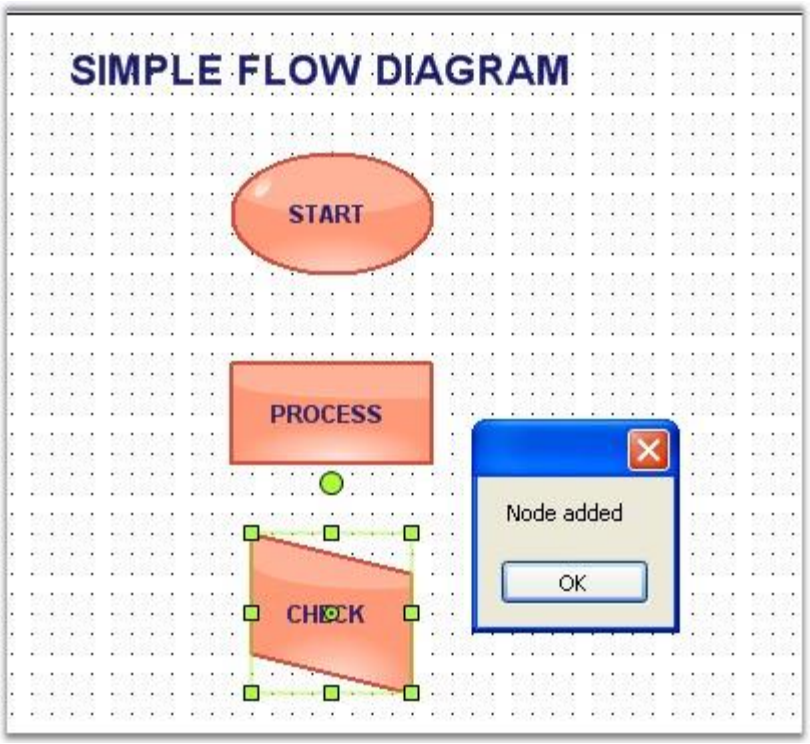

Figure 127: Node collection Changed Event

4.6.8.1.2 Tool Events

The below events gets fired while activating or deactivating the UI tools (Zoom, Pan, Select etc) in the diagram.

The below table shows all the Tool Events.

DiagramViewerEventSink	Description
ToolActivated	Triggered when UI tool is activated.
ToolDeactivated	Triggered when UI tool is deactivated.

Data can be retrieved or set using the following members.

ToolActivated / Deactivated EventArgs Members	Description
Tool	Returns the tools object that generated the event. It has following properties, • Name - Name of the Tool.

In the below code sample, when a tool is activated or deactivated the corresponding event will be raised, and the tool name along with the status will be displayed.

```
[C#]

private void Form1_Load(object sender, EventArgs e)
{
    ((DiagramViewerEventSink)diagram1.EventSink).ToolActivated += new
ToolEventHandler(DiagramForm_ToolActivated);

    ((DiagramViewerEventSink)diagram1.EventSink).ToolDeactivated += new
ToolEventHandler(Form1_ToolDeactivated);

    diagram1.Controller.ActivateTool("ZoomTool");
}

void Form1_ToolDeactivated(ToolEventArgs e)
{
    MessageBox.Show("Deactivated Tool Name: " + e.Tool.Name);
```

```
}
private void DiagramForm_ToolActivated(ToolEventArgs e)
{
    MessageBox.Show("Activated Tool Name: " + e.Tool.Name + "\n" +
"Status: " + e.Tool.InAction);

}
```

```
[VB]

Private Sub Form1_Load(ByVal sender As Object, ByVal e As EventArgs)
    AddHandler DirectCast(diagram1.EventSink,
DiagramViewerEventSink).ToolActivated, AddressOf
DiagramForm_ToolActivated

    AddHandler DirectCast(diagram1.EventSink,
DiagramViewerEventSink).ToolDeactivated, AddressOf
Form1_ToolDeactivated

    diagram1.Controller.ActivateTool("ZoomTool")
End Sub

Private Sub Form1_ToolDeactivated(ByVal e As ToolEventArgs)

    MessageBox.Show("Deactivated Tool Name: " & e.Tool.Name)
End Sub
Private Sub DiagramForm_ToolActivated(ByVal e As ToolEventArgs)

    MessageBox.Show(("Activated Tool Name: " & e.Tool.Name & vbLf &
"Status: ") + e.Tool.InAction)
End Sub
```

Sample diagrams are as follows,

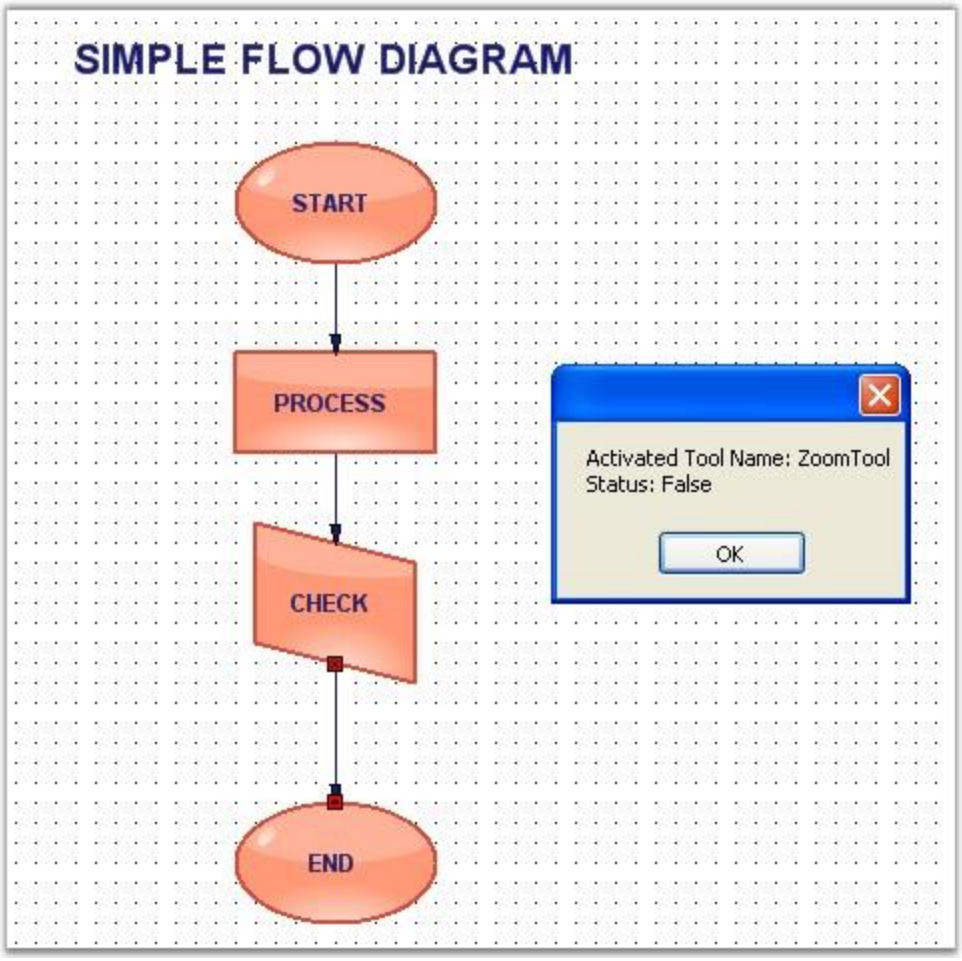

Figure 128: Tool Activated Event

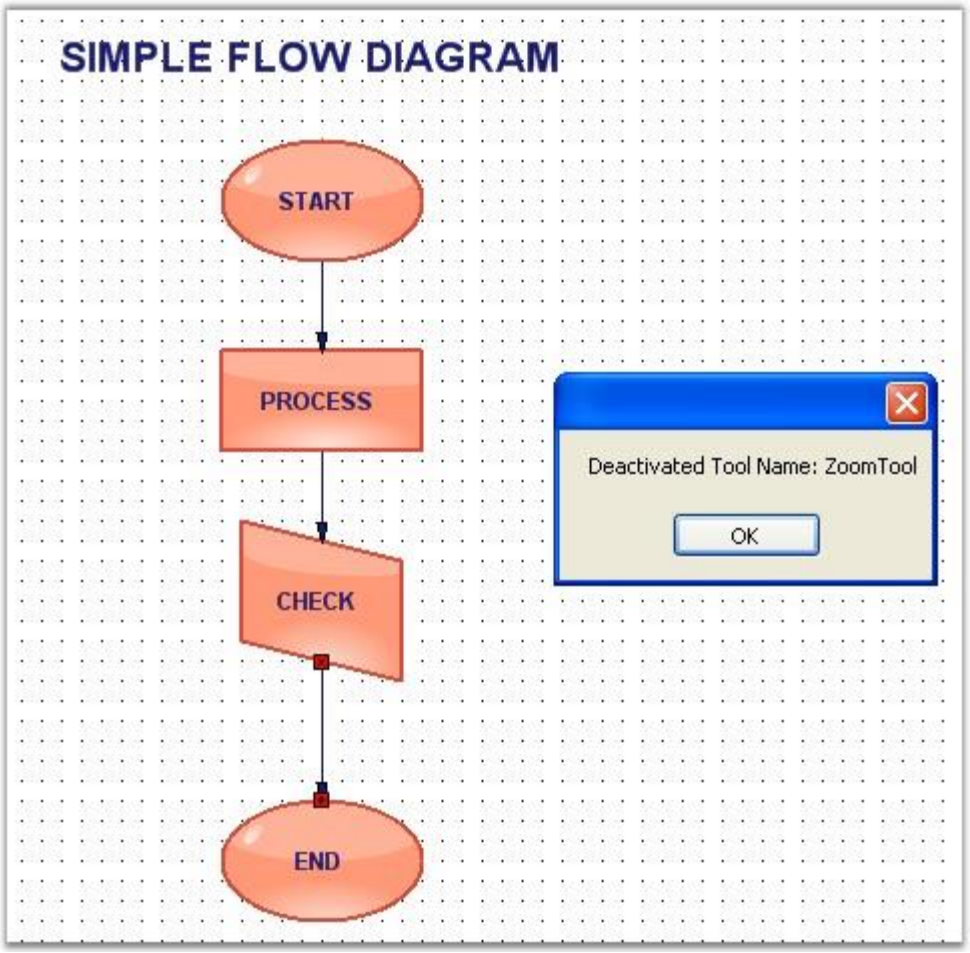

Figure 129: Tool Deactivated Event

4.6.8.1.3 Origin Events

The origin changes when the diagram window is scrolled either horizontally or vertically.

Origin events are as follows,

DiagramViewerEventSink	Description
OriginChanged	Triggered when the origin is changed.

Data can be retrieved or set by using the following members.

Origin EventArgs Member	Description
NewOrigin	Returns the new X and Y origin values after moving the origin.
Offset	Returns the difference between the old and new origin.
OriginalOrigin	Returns the X and Y values before moving the origin.

In the following code sample, when the **OriginChanged** event is handled, the various member values are listed in a text box as the control is scrolled accordingly.

```
[C#]

public void Form1_Load(object sender, EventArgs e)
{

    ((DiagramViewerEventSink)diagram1.EventSink).OriginChanged += new
ViewOriginEventHandler(Form1_OriginChanged);
    NodeCollection nodeStack = new NodeCollection();

    // Circle

    Syncfusion.Windows.Forms.Diagram.Ellipse circle = new
Syncfusion.Windows.Forms.Diagram.Ellipse(0, 0, 96, 72);
    circle.Name = "Circle";
    circle.FillStyle.Type = FillStyleType.LinearGradient;
    circle.FillStyle.ForeColor = Color.AliceBlue;
    circle.ShadowStyle.Visible = true;
    nodeStack.Add(circle);

    // Polygon

    PointF[] pts ={ new Point(6, 36), new Point(48, 6), new Point(90,
36), new Point(48, 66) };
    Polygon polygon = new Polygon(pts);
    polygon.Name = "Polygon";
    polygon.FillStyle.ForeColor = Color.DarkSeaGreen;
    polygon.FillStyle.Color = Color.DarkSeaGreen;
    nodeStack.Add(polygon);

    int i = 0;
    this.model4.AppendChildren(nodeStack, out i);
    MessageBox.Show("Node count =" + "\n" +
```

```
nodeStack.Count.ToString());

    textBox1.Text = model4.Nodes.Last.Name;

    this.diagram1.HScroll = true;
    this.diagram1.VScroll = true;
}

private void Form1_OriginChanged(ViewOriginEventArgs e)
{
    textBox1.Text = "X = " + e.OriginalOrigin.X + "," + "Y = " +
e.OriginalOrigin.Y + "  " + "New X= " + e.NewOrigin.X + "," + "New
      Y= " + e.NewOrigin.Y;

}
```

[VB]

```
Public Sub Form1_Load(ByVal sender As Object, ByVal e As EventArgs)

    AddHandler DirectCast(diagram1.EventSink,
DiagramViewerEventSink).OriginChanged, AddressOf Form1_OriginChanged
    Dim nodeStack As New NodeCollection()

    ' Circle

    Dim circle As New Syncfusion.Windows.Forms.Diagram.Ellipse(0, 0,
96, 72)
    circle.Name = "Circle"
    circle.FillStyle.Type = FillStyleType.LinearGradient
    circle.FillStyle.ForeColor = Color.AliceBlue
    circle.ShadowStyle.Visible = True
    nodeStack.Add(circle)

    ' Polygon

    Dim pts As PointF() = {New Point(6, 36), New Point(48, 6), New
Point(90, 36), New Point(48, 66)}
    Dim polygon As New Polygon(pts)
    polygon.Name = "Polygon"
    polygon.FillStyle.ForeColor = Color.DarkSeaGreen
    polygon.FillStyle.Color = Color.DarkSeaGreen
    nodeStack.Add(polygon)

    Dim i As Integer = 0
    Me.model4.AppendChildren(nodeStack, i)
```



```
    MessageBox.Show(("Node count =" & vbLf) +
nodeStack.Count.ToString())

    textBox1.Text = model4.Nodes.Last.Name

    Me.diagram1.HScroll = True
    Me.diagram1.VScroll = True
End Sub

Private Sub Form1_OriginChanged(ByVal e As ViewOriginEventArgs)

    textBox1.Text = ((("X = " & e.OriginalOrigin.X & "," & "Y = ") +
e.OriginalOrigin.Y & "  " & "New X= ") + e.NewOrigin.X & "," & "New " &
vbTab & "Y= ") + e.NewOrigin.Y
End Sub
```

Sample diagrams are as follows.

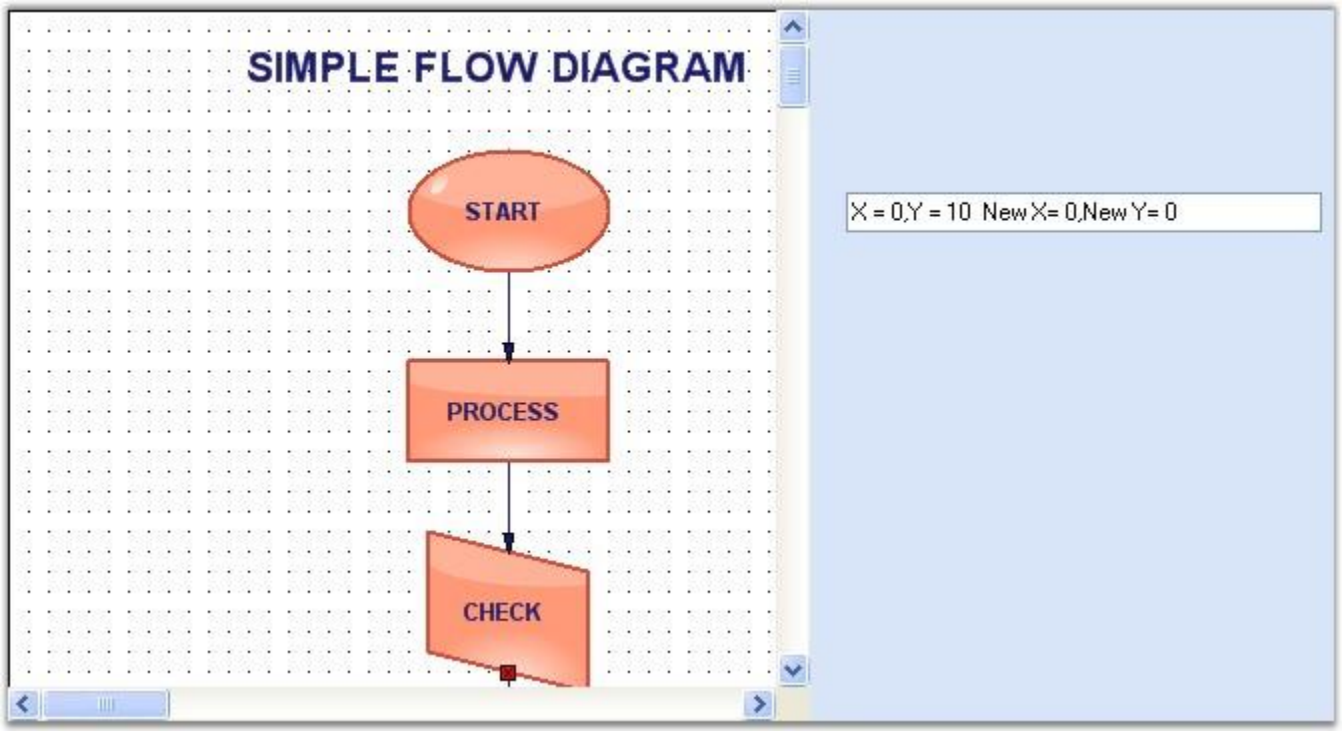

Figure 130: Origin Initial Stage

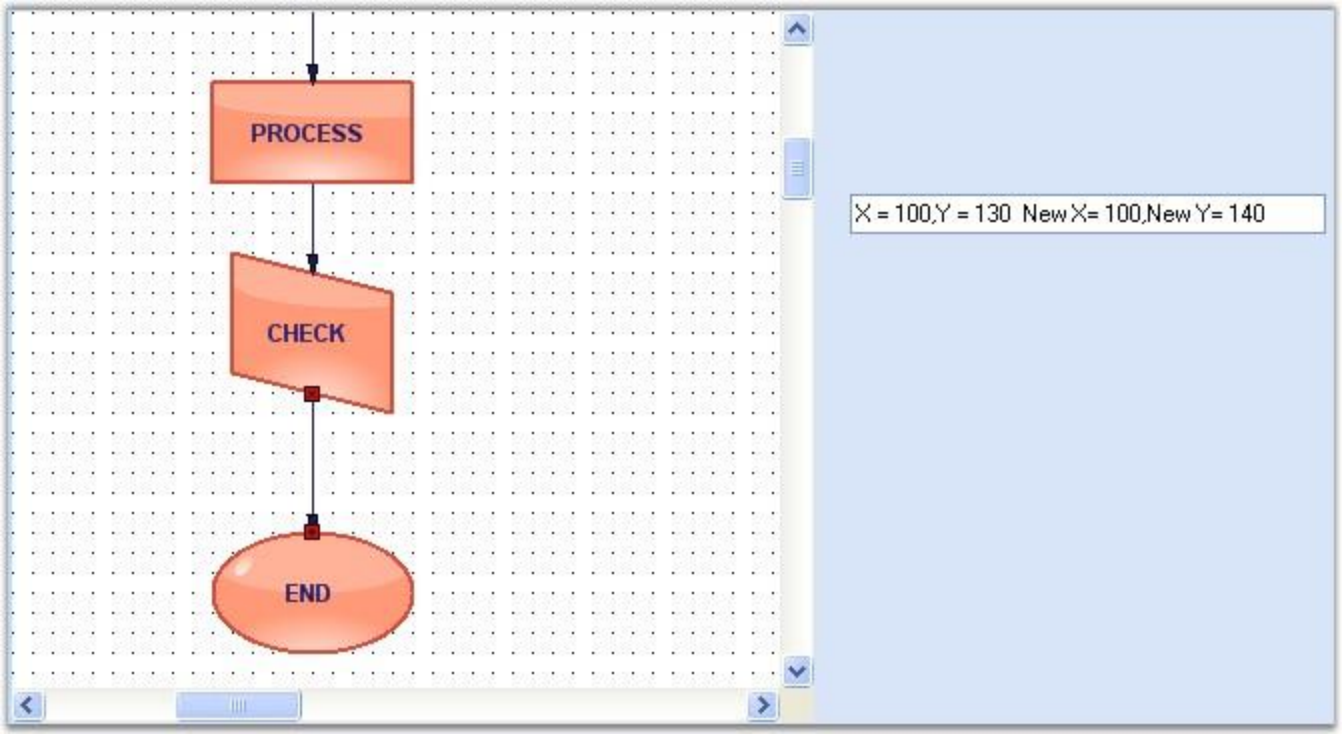

Figure 131: Origin Changed Stage

4.6.8.1.4 Magnification Event

When the control is zoomed in or out, the magnification events will be fired displaying the old and new magnification factors.

Magnification Events are as follows,

DiagramViewerEventSink	Description
MagnificationChanged	Fired when magnification value is changed.

Data can be retrieved or set using the following members.

Magnification EventArgs Member	Description
NewMagnification	Returns the new magnification value.
OriginalMagnification	Returns the old magnification value before the

	event occurred.

[C#]

```
private void Form1_Load(object sender, EventArgs e)
{
    ((DiagramViewerEventSink)diagram1.EventSink).MagnificationChanged
+= new ViewMagnificationEventHandler(Form1_MagnificationChanged);
}

[EventHandlerPriorityAttribute(true)]
private void Form1_MagnificationChanged(ViewMagnificationEventArgs
evtArgs)
{
    MessageBox.Show("Old Factor: " +
evtArgs.OriginalMagnification.ToString() + "New Factor: " +
evtArgs.NewMagnification.ToString());
}
```

[VB]

```
Private Sub Form1_Load(ByVal sender As Object, ByVal e As EventArgs)
    AddHandler DirectCast(diagram1.EventSink,
DiagramViewerEventSink).MagnificationChanged, AddressOf
Form1_MagnificationChanged
End Sub

<EventHandlerPriorityAttribute(True)> _
Private Sub Form1_MagnificationChanged(ByVal evtArgs As
ViewMagnificationEventArgs)
    MessageBox.Show(("Old Factor: " &
evtArgs.OriginalMagnification.ToString() & "New Factor: ") +
evtArgs.NewMagnification.ToString())
End Sub
```

Sample diagrams are as follows,

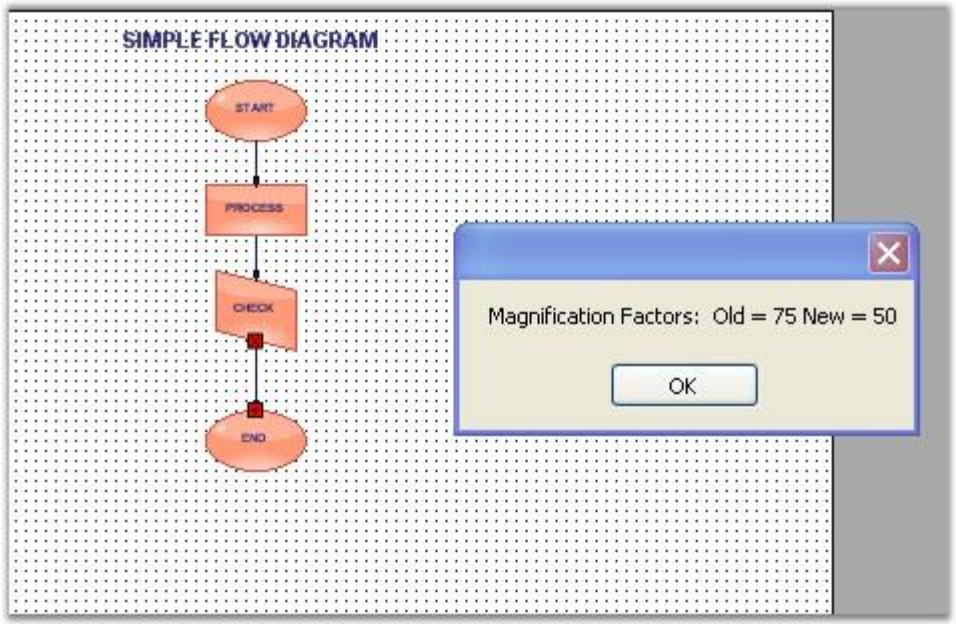

Figure 132: Magnification Factor while Zoom In

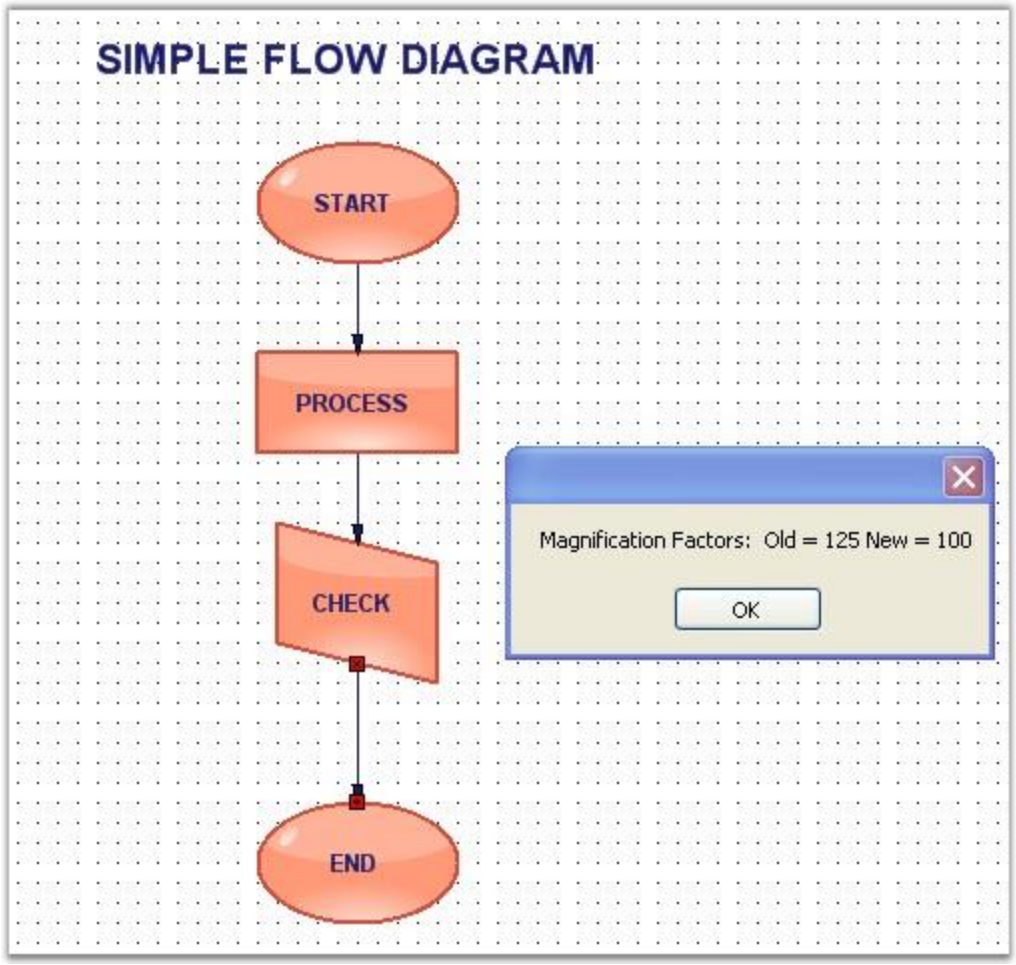

Figure 133: Magnification Factor while Zoom Out

4.6.8.2 Model Events

DocumentEventSink class contains the events specific to the document. The various events that can be invoked using this class are discussed in the following topics.

4.6.8.2.1 Vertex Events

Vertex Events are as follows,

DocumentEventSink	Description
VertexChanged	Gets fired after the vertex of the node has been changed.
VertexChanging	Gets fired when the vertex of the node is changed.

Data can be retrieved or set using the following members.

Vertex EventArgs Member	Description
Cancel	Cancels the Vertex Changed event from being fired.
ChangeType	It returns the following possible value: • Set - whether the vertex is set for the node.
NodeAffected	Returns the node's name by which the node was affected.
VertexIndex	Returns the index of the current vertex.
VertexLocation	Returns the position of the vertex.

Programmatically the events are written as follows,

```
[C#]

public void Form1_Load(object sender, EventArgs e)
{
    ((DocumentEventSink)model1.EventSink).VertexChanged += new
VertexChangedEventHandler(Form1_VertexChanged);
    ((DocumentEventSink)model1.EventSink).VertexChanging += new
VertexChangingEventHandler(Form1_VertexChanging);
    LineConnector line = new LineConnector(circle.PinPoint,
polygon.PinPoint);
    polygon.CentralPort.TryConnect(line.HeadEndPoint);
    circle.CentralPort.TryConnect(line.TailEndPoint);
    model1.AppendChild(line);

}
```



```
private void Form1_VertexChanging(VertexChangingEventArgs vertexChange)
{

    MessageBox.Show("VertexChanging fired");
    model1.LineStyle.LineWidth = 2;

}

private void Form1_VertexChanged(VertexChangedEventArgs vertexChange)
{

    MessageBox.Show("Target Node - " +
vertexChange.NodeAffected.FullName + "\n" +
vertexChange.VertexLocation.ToString());

}
```

```
[VB]

Public Sub Form1_Load(ByVal sender As Object, ByVal e As EventArgs)
    AddHandler DirectCast(model1.EventSink,
DocumentEventSink).VertexChanged, AddressOf Form1_VertexChanged
    AddHandler DirectCast(model1.EventSink,
DocumentEventSink).VertexChanging, AddressOf Form1_VertexChanging
    Dim line As New LineConnector(circle.PinPoint, polygon.PinPoint)
    polygon.CentralPort.TryConnect(line.HeadEndPoint)
    circle.CentralPort.TryConnect(line.TailEndPoint)

    model1.AppendChild(line)
End Sub

Private Sub Form1_VertexChanging(ByVal vertexChange As
VertexChangingEventArgs)

    MessageBox.Show("VertexChanging fired")

    model1.LineStyle.LineWidth = 2
End Sub

Private Sub Form1_VertexChanged(ByVal vertexChange As
VertexChangedEventArgs)


    MessageBox.Show(("Target Node - " &
vertexChange.NodeAffected.FullName & vbLf) +
vertexChange.VertexLocation.ToString())
End Sub
```


Sample diagrams are as follows,

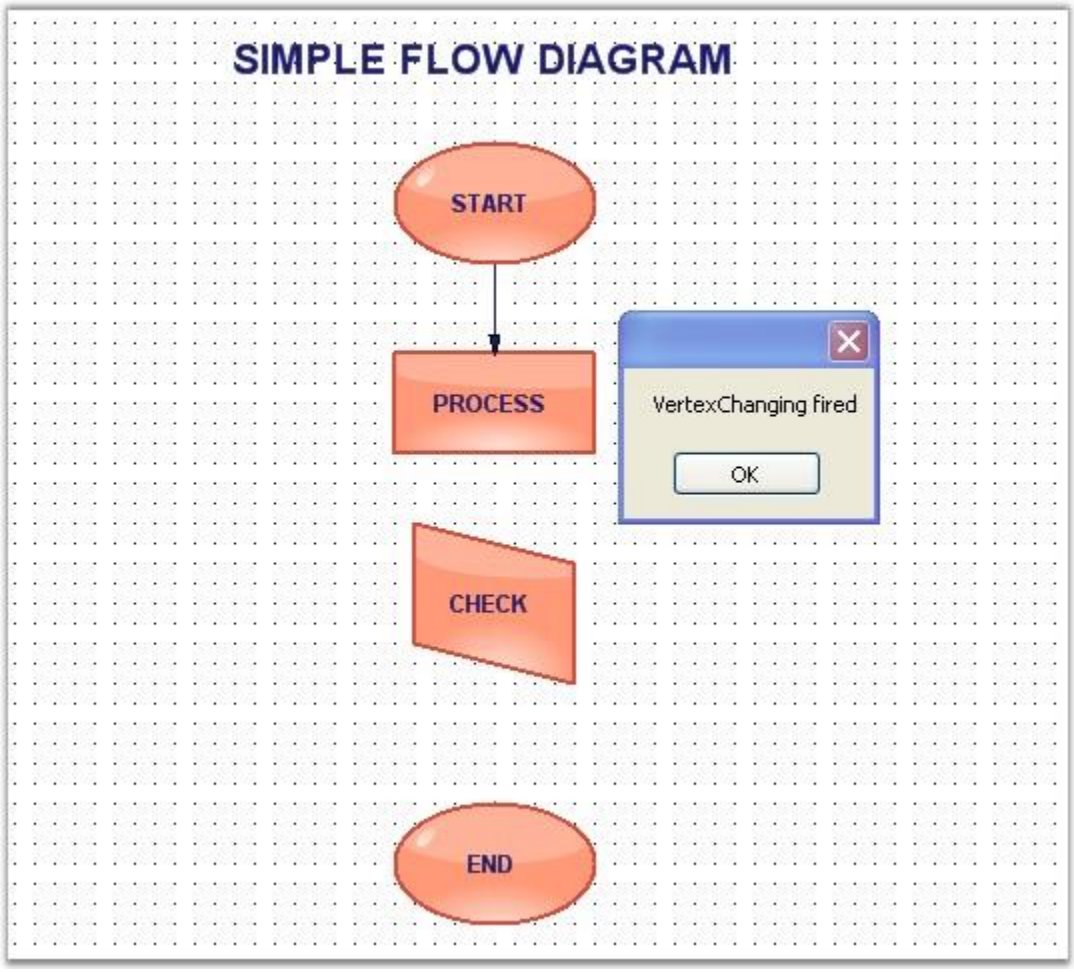

Figure 134: Diagram with Vertex Changing Event

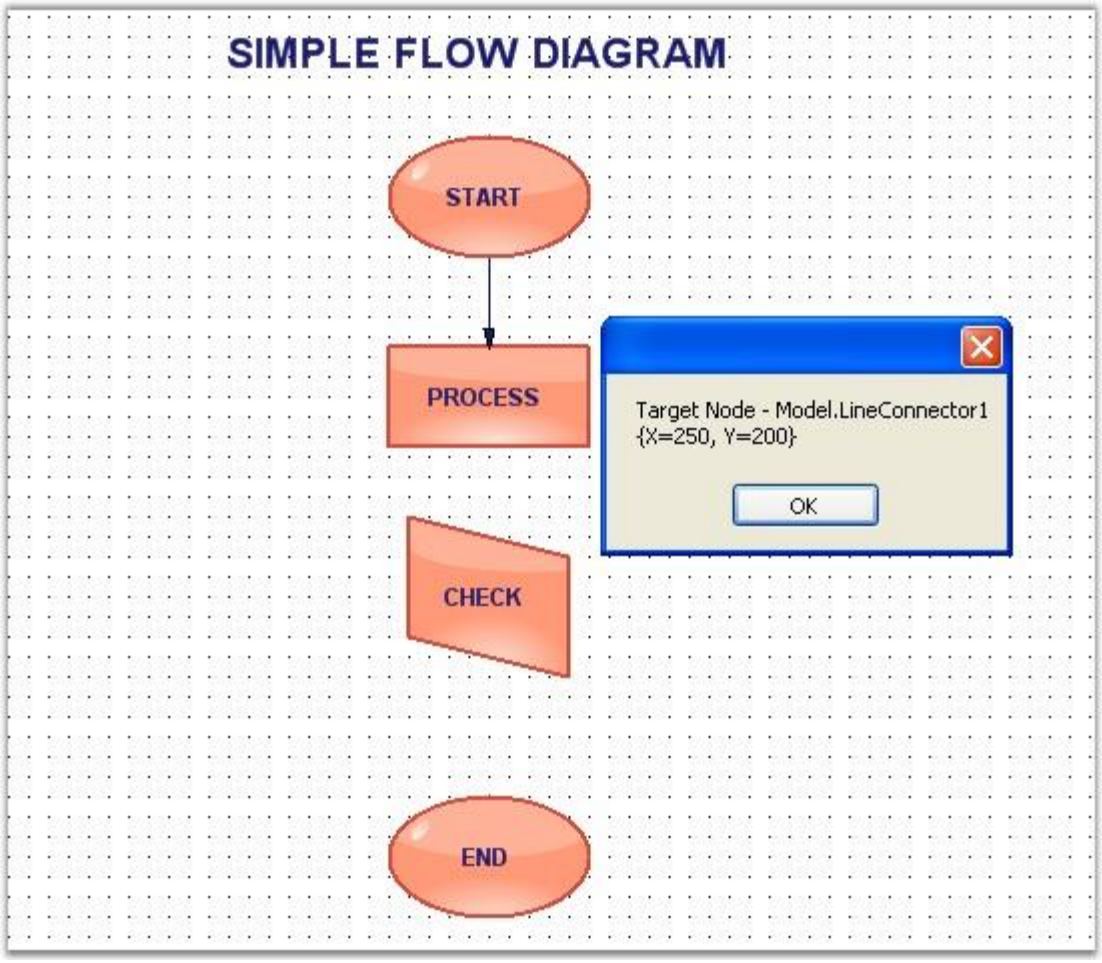

Figure 135: Diagram with Vertex Changed Event

4.6.8.2.2 PinPoint Events

When changing the node's location, the pinpoint of the node will be reset. The below table contains pinpoint events and descriptions.

DocumentEventSink	Description
PinOffsetChanged	Triggered after the offset of the pinpoint is reset.
PinOffsetChanging	Triggered when the offset of the pinpoint is changed.

PinPointChanged	Triggered after the pinpoint is repositioned.
PinPointChanging	Triggered when the pinpoint is moved.

Data can be retrieved or set using the following members.

PinPoint / PinPointOffset EventArgs Member	Description
Cancel	Cancels the PinPoint Changing events.
NodeAffected	Returns the node's name by which the node was affected.
Offset	Returns the X and Y values.

Programmatically the events are written as follows,

```
[C#]

public void Form1_Load(object sender, EventArgs e)
{

    ((DocumentEventSink)model4.EventSink).PinOffsetChanged += new
PinOffsetChangedEventHandler(Form1_PinOffsetChanged);
    ((DocumentEventSink)model4.EventSink).PinOffsetChanging += new
PinOffsetChangingEventHandler(Form1_PinOffsetChanging);
    ((DocumentEventSink)model4.EventSink).PinPointChanged += new
PinPointChangedEventHandler(Form1_PinPointChanged);
    ((DocumentEventSink)model4.EventSink).PinPointChanging += new
PinPointChangingEventHandler(Form1_PinPointChanging);

    // Circle
    Syncfusion.Windows.Forms.Diagram.Ellipse circle = new
Syncfusion.Windows.Forms.Diagram.Ellipse(0, 0, 96, 72);
    circle.Name = "Circle";
    circle.FillStyle.Type = FillStyleType.LinearGradient;
    circle.FillStyle.ForeColor = Color.AliceBlue;
    circle.ShadowStyle.Visible = true;
    model4.AppendChild(circle);
}

void Form1_PinPointChanging(PinPointChangingEventArgs evtArgs)
{
    MessageBox.Show("PinpointChanging event is fired" + "\n" + "Node
```

```
name: " + evtArgs.NodeAffected.Name.ToString());
}

void Form1_PinPointChanged(PinPointChangedEventArgs evtArgs)
{
    MessageBox.Show("PinPointChanged event is fired" + "\n" + "Offset
values: " + evtArgs.Offset.ToString());
}

void Form1_PinOffsetChanging(PinOffsetChangingEventArgs evtArgs)
{
    MessageBox.Show("PinOffsetChanging event is fired" + "\n" + "Node
name: " + evtArgs.NodeAffected.Name);
}

void Form1_PinOffsetChanged(PinOffsetChangedEventArgs evtArgs)
{
    MessageBox.Show("PinOffsetChanged event is fired" + "\n" + "Offset
values: " + evtArgs.Offset.ToString());
}
```

[VB]

```
Public Sub Form1_Load(ByVal sender As Object, ByVal e As EventArgs)

    AddHandler DirectCast(model4.EventSink,
DocumentEventSink).PinOffsetChanged, AddressOf Form1_PinOffsetChanged
    AddHandler DirectCast(model4.EventSink,
DocumentEventSink).PinOffsetChanging, AddressOf Form1_PinOffsetChanging
    AddHandler DirectCast(model4.EventSink,
DocumentEventSink).PinPointChanged, AddressOf Form1_PinPointChanged
    AddHandler DirectCast(model4.EventSink,
DocumentEventSink).PinPointChanging, AddressOf Form1_PinPointChanging

    ' Circle
    Dim circle As New Syncfusion.Windows.Forms.Diagram.Ellipse(0, 0,
96, 72)
    circle.Name = "Circle"
    circle.FillStyle.Type = FillStyleType.LinearGradient
    circle.FillStyle.ForeColor = Color.AliceBlue
    circle.ShadowStyle.Visible = True
    model4.AppendChild(circle)
End Sub

Private Sub Form1_PinPointChanging(ByVal evtArgs As
PinPointChangingEventArgs)
    MessageBox.Show(("PinpointChanging event is fired" & vbLf & "Node
```



```
name: ") + evtArgs.NodeAffected.Name.ToString())
End Sub

Private Sub Form1_PinPointChanged(ByVal evtArgs As
PinPointChangedEventArgs)
    MessageBox.Show(("PinPointChanged event is fired" & vbLf & "Offset
values: ") + evtArgs.Offset.ToString())
End Sub

Private Sub Form1_PinOffsetChanging(ByVal evtArgs As
PinOffsetChangingEventArgs)
    MessageBox.Show(("PinOffsetChanging event is fired" & vbLf & "Node
name: ") + evtArgs.NodeAffected.Name)
End Sub

Private Sub Form1_PinOffsetChanged(ByVal evtArgs As
PinOffsetChangedEventArgs)
    MessageBox.Show(("PinOffsetChanged event is fired" & vbLf & "Offset
values: ") + evtArgs.Offset.ToString())
End Sub
```

Sample diagrams are as follows,

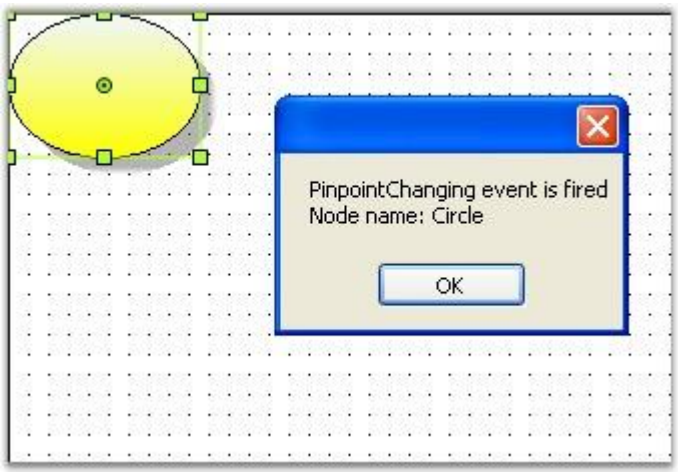

Figure 136: Pinpoint Changing Event

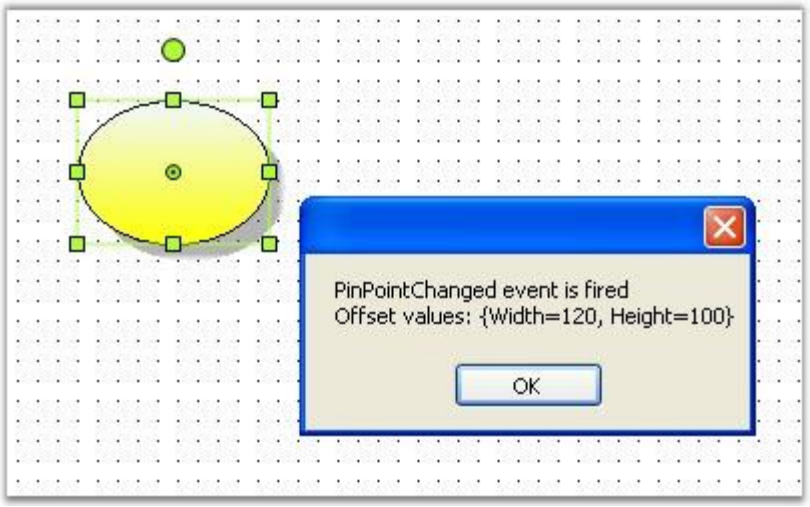

Figure 137: Pinpoint Changed Event

4.6.8.2.3 Rotation Events

When the control is rotated horizontally or vertically, the rotation events will be fired displaying the rotation offsets.

The below table discusses the available rotation events with descriptions.

DocumentEventSink	Description
FlipChanged	Triggered after the node is rotated using Flip property.
FlipChanging	Triggered when the node is rotated using Flip property.
RotationChanged	Triggered after the node is rotated.
RotationChanging	Triggered on rotating the node in any direction.

Data can be retrieved or set using the following members.

Rotation EventArgs Member	Description

NodeAffected	Returns the node's name by which the node was affected.
RotationOffset	Returns the angle by which the node was rotated.

Flip EventArgs Member	Description
Cancel	Cancels the FlipChanging event.
FlipAxis	Returns the axis around which the node was rotated.
FlipValue	Returns the boolean value of the **Flip** property.
NodeAffected	Returns the node's name by which the node was affected.

Programmatically, the events are written as follows:

```
[C#]

public void Form1_Load(object sender, EventArgs e)
{
    ((DocumentEventSink)model1.EventSink).FlipChanged += new
FlipChangedEventHandler(Form1_FlipChanged);
    ((DocumentEventSink)model1.EventSink).FlipChanging += new
FlipChangingEventHandler(Form1_FlipChanging);
    ((DocumentEventSink)model1.EventSink).RotationChanged += new
RotationChangedEventHandler(Form1_RotationChanged);
    ((DocumentEventSink)model1.EventSink).RotationChanging += new
RotationChangingEventHandler(Form1_RotationChanging);
    // Circle
    Syncfusion.Windows.Forms.Diagram.Ellipse circle = new
Syncfusion.Windows.Forms.Diagram.Ellipse(0, 0, 96, 72);
    circle.Name = "Circle";
    circle.FillStyle.Type = FillStyleType.LinearGradient;
    circle.FillStyle.ForeColor = Color.AliceBlue;
    circle.ShadowStyle.Visible = true;
    model4.AppendChild(circle);
}
void Form1_RotationChanged(RotationChangedEventArgs evtArgs)
{
    MessageBox.Show("RotationChanged event is fired" + "\n" +
evtArgs.RotationOffset.ToString());
}
```

```
void Form1_FlipChanging(FlipChangingEventArgs evtArgs)
{
    MessageBox.Show("FlipChanging event is fired");
    textBox1.Text = evtArgs.NodeAffected.BoundingRectangle.ToString();
}
void Form1_FlipChanged(FlipChangedEventArgs evtArgs)
{
    MessageBox.Show("FlipChanged event is fired" + "\n" + "Flip Axis:"
+ evtArgs.FlipAxis.ToString() + "\n" + "Node: " +
evtArgs.NodeAffected.Name.ToString());
    evtArgs.NodeAffected.EditStyle.Enabled = false;
}
```

[VB]

```
Public Sub Form1_Load(ByVal sender As Object, ByVal e As EventArgs)
    AddHandler DirectCast(model1.EventSink,
DocumentEventSink).FlipChanged, AddressOf Form1_FlipChanged
    AddHandler DirectCast(model1.EventSink,
DocumentEventSink).FlipChanging, AddressOf Form1_FlipChanging
    AddHandler DirectCast(model1.EventSink,
DocumentEventSink).RotationChanged, AddressOf Form1_RotationChanged
    AddHandler DirectCast(model1.EventSink,
DocumentEventSink).RotationChanging, AddressOf Form1_RotationChanging
    ' Circle
    Dim circle As New Syncfusion.Windows.Forms.Diagram.Ellipse(0, 0,
96, 72)
    circle.Name = "Circle"
    circle.FillStyle.Type = FillStyleType.LinearGradient
    circle.FillStyle.ForeColor = Color.AliceBlue
    circle.ShadowStyle.Visible = True
    model4.AppendChild(circle)
End Sub
Private Sub Form1_RotationChanged(ByVal evtArgs As
RotationChangedEventArgs)
    MessageBox.Show(("RotationChanged event is fired" & vbLf) +
evtArgs.RotationOffset.ToString())
End Sub
Private Sub Form1_FlipChanging(ByVal evtArgs As FlipChangingEventArgs)
    MessageBox.Show("FlipChanging event is fired")
    textBox1.Text = evtArgs.NodeAffected.BoundingRectangle.ToString()
End Sub
Private Sub Form1_FlipChanged(ByVal evtArgs As FlipChangedEventArgs)
    MessageBox.Show((("FlipChanged event is fired" & vbLf & "Flip
Axis:") + evtArgs.FlipAxis.ToString() & vbLf & "Node: ") +
evtArgs.NodeAffected.Name.ToString())
    evtArgs.NodeAffected.EditStyle.Enabled = False
```



```
End Sub
```

Sample diagrams are as follows:

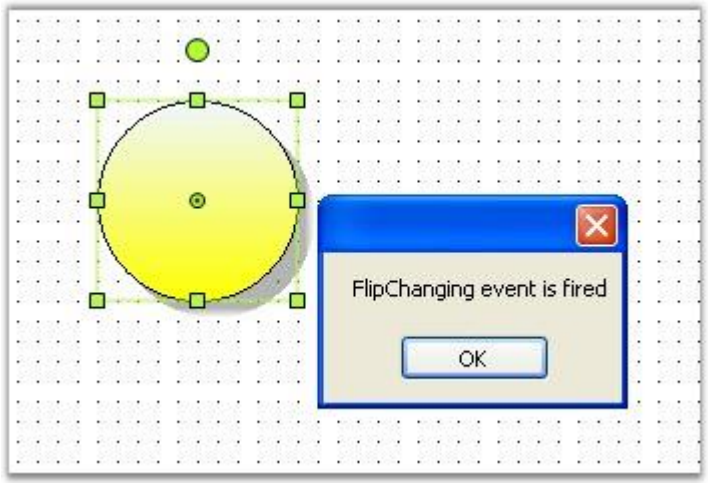

Figure 138: Flip Changing Event

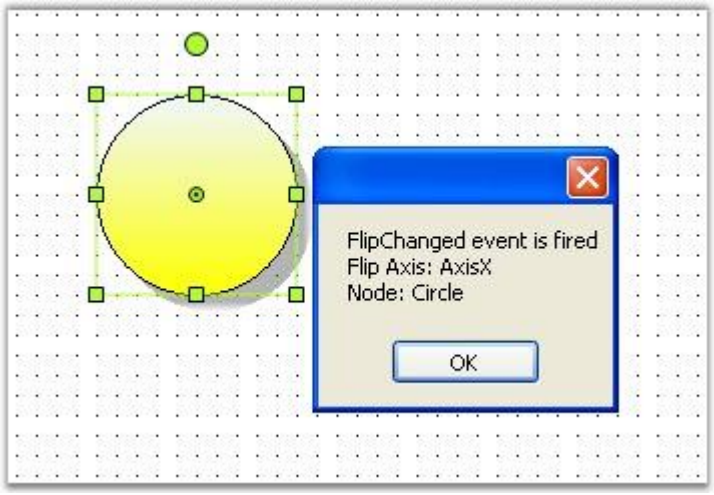

Figure 139: Flip Changed Event

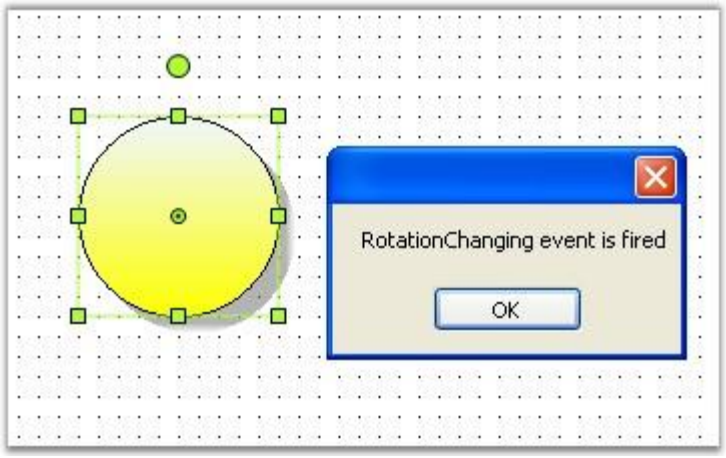

Figure 140: Rotation Changing Event

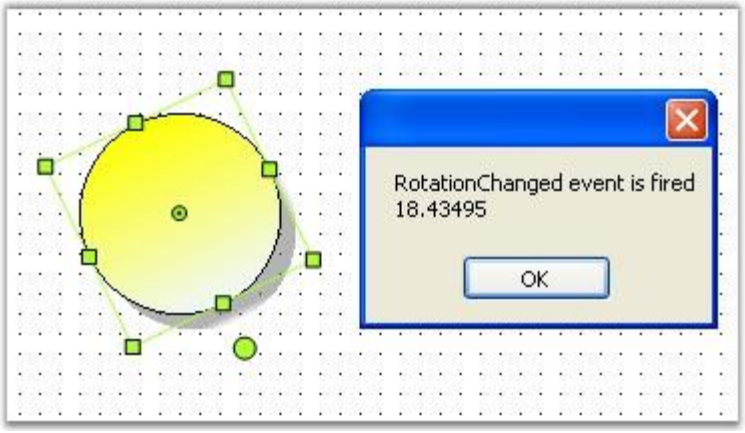

Figure 141: Rotation Changed Event

4.6.8.2.4 Z-Order Events

When the node order is changed from front-to-back or back-to-front, the Z-Order value gets changed and triggers the ZOrder events listed in the below table.

DocumentEventSink	Description
ZOrderChanged	Gets fired after the ZOrder value is changed.
ZOrderChanging	Gets fired when the ZOrder of the node is changed.

Data can be retrieved / set by using the following members.

ZOrder EventArgs Members	Description
Cancel	Cancels the ZOderChanging event.
ChangeType	It returns the following possible values, • Front-whether the controller bring the node to the front • Back-whether the controller send the node to the back
NodeAffected	Returns the node's name by which the node was affected.
ZOrder	Returns the current zorder value.

Programmatically, the events are written as follows:

[C#]

```
public void Form1_Load(object sender, EventArgs e)
{
    ((DocumentEventSink)model1.EventSink).ZOrderChanged += new
ZOrderChangedEventHandler(Form1_ZOrderChanged);
    ((DocumentEventSink)model1.EventSink).ZOrderChanging += new
ZOrderChangingEventHandler(Form1_ZOrderChanging);

    diagram1.Controller.BringToFront();
}
void Form1_ZOrderChanging(ZOrderChangingEventArgs evtArgs)
{
    MessageBox.Show("ZOrderChanging event is fired" + "\n" + "Node: " +
evtArgs.NodeAffected.Name.ToString());
}
void Form1_ZOrderChanged(ZOrderChangedEventArgs evtArgs)
{
    MessageBox.Show("ZOrderChanged event is fired" + "\n" + "New
ZOrder: " + evtArgs.ZOrder.ToString());
}
```

[VB]

```
Public Sub Form1_Load(ByVal sender As Object, ByVal e As EventArgs)
```

```
    AddHandler DirectCast(model1.EventSink,
DocumentEventSink).ZOrderChanged, AddressOf Form1_ZOrderChanged
    AddHandler DirectCast(model1.EventSink,
DocumentEventSink).ZOrderChanging, AddressOf Form1_ZOrderChanging

    diagram1.Controller.BringToFront()
End Sub
Private Sub Form1_ZOrderChanging(ByVal evtArgs As
ZOrderChangingEventArgs)
   MessageBox.Show(("ZOrderChanging event is fired" & vbLf & "Node: ")
+ evtArgs.NodeAffected.Name.ToString())
End Sub
Private Sub Form1_ZOrderChanged(ByVal evtArgs As
ZOrderChangedEventArgs)
    MessageBox.Show(("ZOrderChanged event is fired" & vbLf & "New
ZOrder: ") + evtArgs.ZOrder.ToString())
End Sub
```

Sample diagram are as follows:

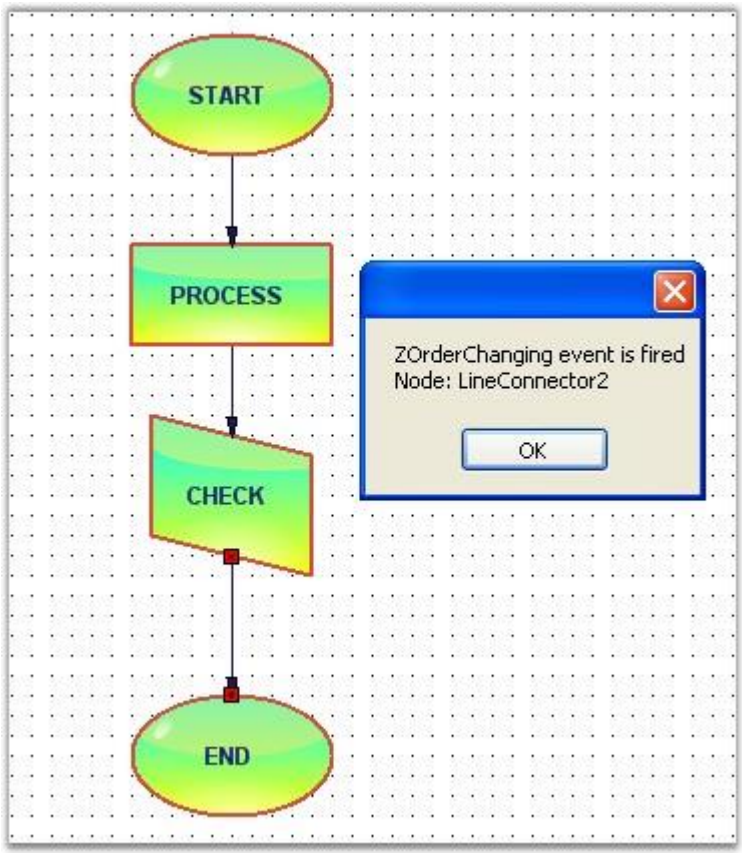

Figure 142: Z-Order Changing Event

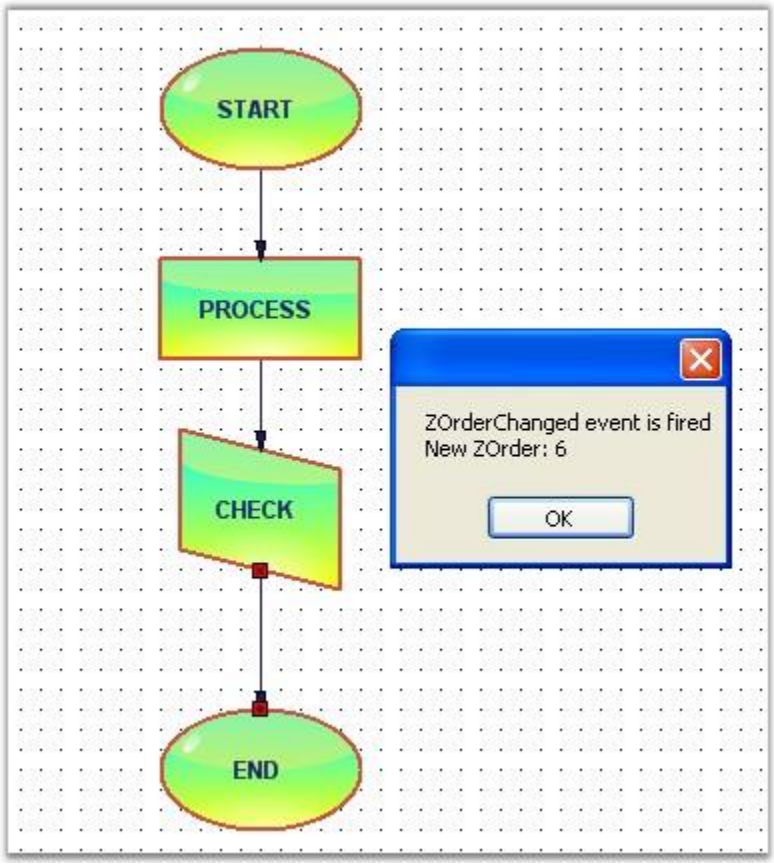

Figure 143: Z-Order Changed Event

4.6.8.2.5 Connections And Ports Events

The below events gets fired while the connection is created between two nodes.

The below table explains the Connections and Ports events.

DocumentEventSink	Description
ConnectionsChanged	Triggered after the connection is changed.
PortsChanged	Triggered when ports are added or changed.

Data can be retrieved or set using the following members.

Connection and Port EventArgs Member	Description
Cancel	Cancels the ConnectionChanging event.
ChangeType	It returns the following possible values: • Insert-Whether the node is inserted • Remove–Whether the node is removed
Element	Returns whether the head or tail end is moved.
Elements	Returns the elements collection on which the event occurs.
Index	Returns the zero-based index into the collection on which the event occurred.
Owner	Returns the owner object. This is a read-only boolean value.

Connection Events

Programmatically, the Connection Event is handled as follows.

[C#]

```
public void Form1_Load(object sender, EventArgs e)
{

    ((DocumentEventSink)model1.EventSink).ConnectionsChanged += new
CollectionExEventHandler(Form1_ConnectionsChanged);
    LineConnector line = new LineConnector(circle.PinPoint,
polygon.PinPoint);
    polygon.CentralPort.TryConnect(line.HeadEndPoint);
    circle.CentralPort.TryConnect(line.TailEndPoint);
    model1.AppendChild(line);
}
void Form1_ConnectionsChanged(CollectionExEventArgs evtArgs)
{
    MessageBox.Show(evtArgs.ChangeType.ToString());
}
```

[VB]

```
Public Sub Form1_Load(ByVal sender As Object, ByVal e As EventArgs)
    AddHandler DirectCast(model1.EventSink,
DocumentEventSink).ConnectionsChanged, AddressOf
Form1_ConnectionsChanged
    Dim line As New LineConnector(circle.PinPoint, polygon.PinPoint)
    polygon.CentralPort.TryConnect(line.HeadEndPoint)
    circle.CentralPort.TryConnect(line.TailEndPoint)
    model1.AppendChild(line)
End Sub
Private Sub Form1_ConnectionsChanged(ByVal evtArgs As
CollectionExEventArgs)
    MessageBox.Show(evtArgs.ChangeType.ToString())
End Sub
```

Sample diagram is as follows.

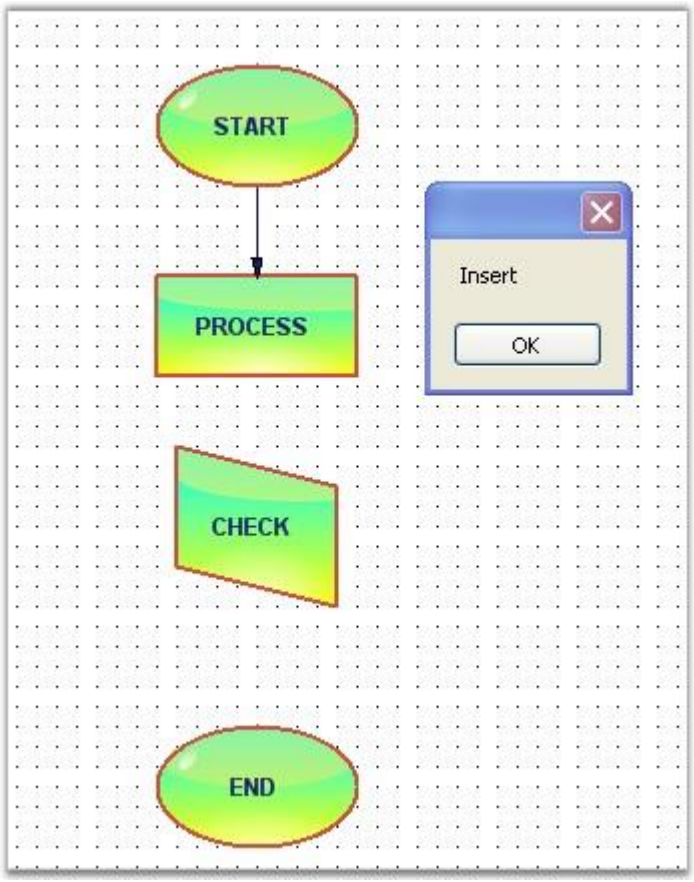

Figure 144: Connection Changed Event

Ports Events

Programmatically, the events are handled as follows.

[C#]

```
public void Form1_Load(object sender, EventArgs e)
{
    ((DocumentEventSink)model1.EventSink).PortsChanged += new
CollectionExEventHandler(Form1_PortsChanged);
    node.EnableCentralPort = false;
}

void Form1_PortsChanged(CollectionExEventArgs evtArgs)
{
    MessageBox.Show("Port is changed");
}
```

[VB]

```
Public Sub Form1_Load(ByVal sender As Object, ByVal e As EventArgs)
    AddHandler DirectCast(model1.EventSink,
DocumentEventSink).PortsChanged, AddressOf Form1_PortsChanged
    node.EnableCentralPort = False
End Sub

Private Sub Form1_PortsChanged(ByVal evtArgs As CollectionExEventArgs)
    MessageBox.Show("Port is changed")
End Sub
```

Sample diagram is as follows.

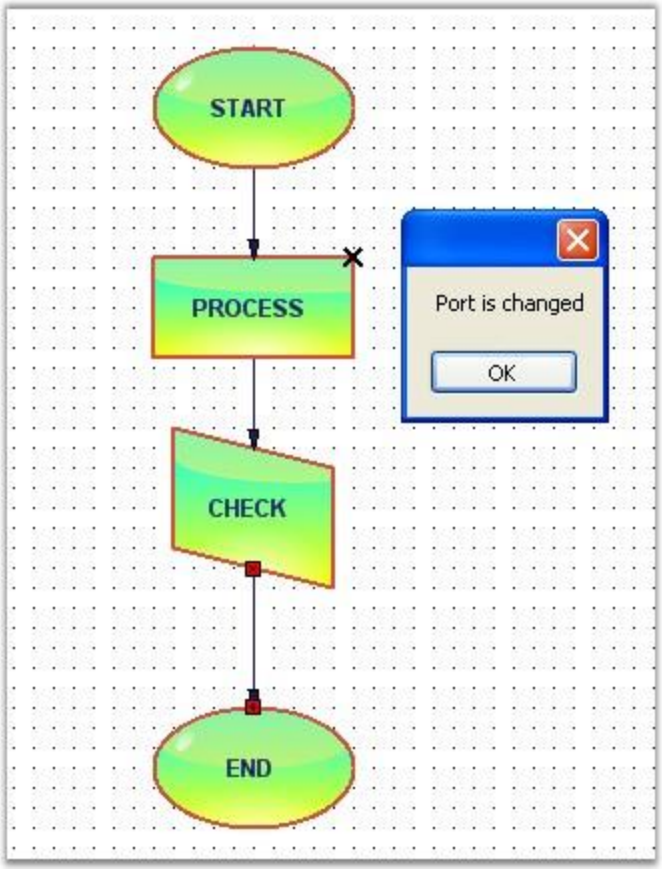

Figure 145: Port Changed Event

4.6.8.2.6 Property Events

Each node has different properties (Name,Color,Size etc). The below events are handled when changing these properties.

Property Events are as follows.

DocumentEventSink	Description
PropertyChanged	Triggered after the property of any node is changed.
PropertyChanging	Triggered when the property value is changed.

Data can be retrieved or set using the following members.

PropertyChanging EventArgs Member	Description
Cancel	Cancels the PropertyChanged event.
NewValue	Returns the new value assigned to the property.
PropertyContainer	Returns the container for the property.
PropertyName	Returns name of the property whose value is changed.

PropertyChanged EventArgs Member	Description
NodeAffected	Returns the name of the node whose property is changed.
PropertyName	Returns the name of the property whose value is changed.

Programmatically the events are written as follows,

```
[C#]

public void Form1_Load(object sender, EventArgs e)
{
    ((DocumentEventSink)model1.EventSink).PropertyChanged += new
Syncfusion.Windows.Forms.Diagram.PropertyChangedEventHandler
        (Form1_PropertyChanged);
    ((DocumentEventSink)model1.EventSink).PropertyChanging += new
PropertyChangingEventHandler(Form1_PropertyChanging);
}
private void
Form1_PropertyChanged(Syncfusion.Windows.Forms.Diagram.PropertyChangedE
ventArgs evtArgs)
{
    MessageBox.Show("PropertyChanged event is fired" + "\n" + "Property
Name: " + evtArgs.PropertyName);
}
private void
Form1 PropertyChanging(Syncfusion.Windows.Forms.Diagram.PropertyChangin
gEventArgs eprop)
{
    MessageBox.Show("PropertyChanging event is fired" + "\n" +
"Property Name: " + eprop.PropertyName + "\n" + "new
```

```
      Value: " + eprop.NewValue);
}
```

```
[VB]

Public Sub Form1_Load(ByVal sender As Object, ByVal e As EventArgs)
    AddHandler DirectCast(model1.EventSink,
DocumentEventSink).PropertyChanged, AddressOf Form1_PropertyChanged
    AddHandler DirectCast(model1.EventSink,
DocumentEventSink).PropertyChanging, AddressOf Form1_PropertyChanging
End Sub
Private Sub Form1_PropertyChanged(ByVal evtArgs As
Syncfusion.Windows.Forms.Diagram.PropertyChangedEventArgs)
    MessageBox.Show(("PropertyChanged event is fired" & vbLf &
"Property Name: ") + evtArgs.PropertyName)
End Sub
Private Sub Form1_PropertyChanging(ByVal eprop As
Syncfusion.Windows.Forms.Diagram.PropertyChangingEventArgs)
    MessageBox.Show((("PropertyChanging event is fired" & vbLf &
"Property Name: ") + eprop.PropertyName & vbLf & "new " & vbTab & vbTab
& vbTab & vbTab & "Value: ") + eprop.NewValue)
End Sub
```

Sample diagrams are as follows.

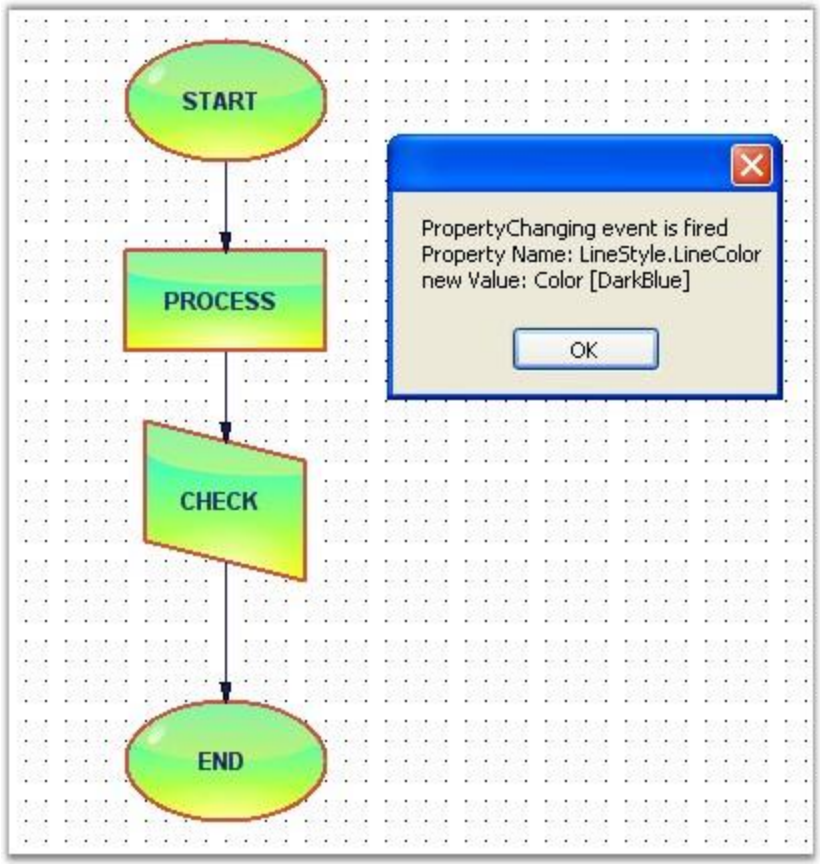

Figure 146: Property Changing Event

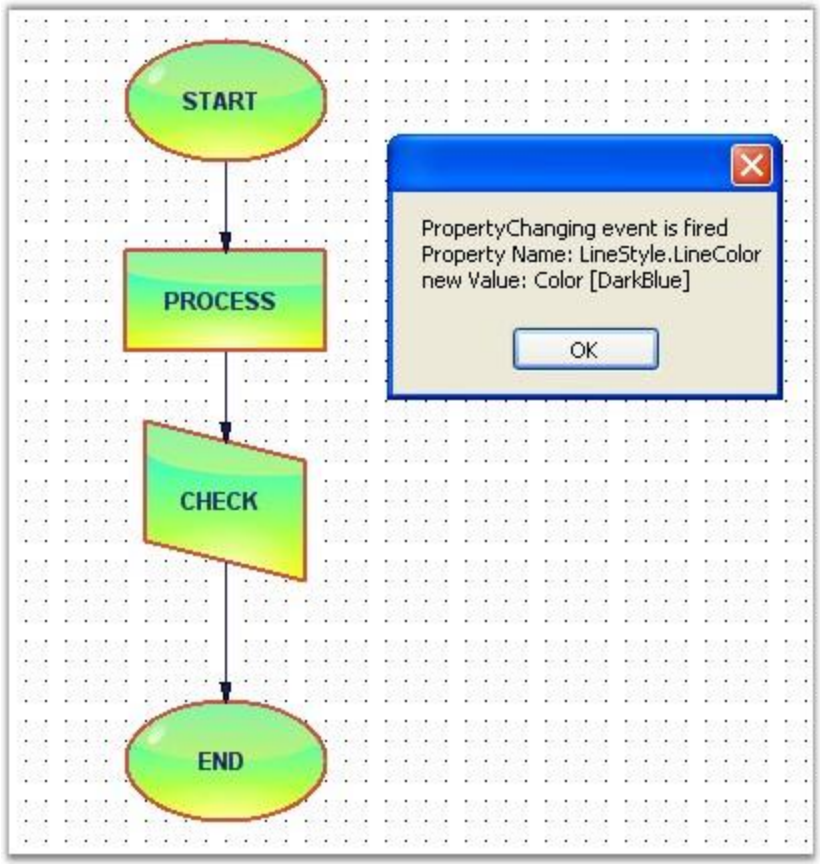

Figure 147: Property Changed Event

4.6.8.2.7 Labels And Layers Events

The below mentioned events are fired, when adding or removing the labels and layers to or from the diagram.

The following table shows the label events:

DocumentEventSink	Description
LabelsChanged	Triggered when labels are added.
LayersChanged	Triggered when layers are added to the model.

Data can be retrieved or set using the following members.

Label / Layers EventArgs Member	Description
Cancel	Cancels the LabelChanging event.
ChangeType	It returns the following possible values: • Insert-Whether the label is inserted • Remove–Whether the label is removed
Element	Returns whether the head or tail end is moved.
Elements	Returns the elements collection on which the event occurs.
Index	Returns the zero-based index into the collection on which the event occurred.
Owner	Returns the owner object.

Label Events

Whenever labels are added to the label collection, this event will be triggered.

Programmatically, the events are written as follows:

[C#]

```
public void Form1_Load(object sender, EventArgs e)
{
    ((DocumentEventSink)model1.EventSink).LabelsChanged += new
CollectionExEventHandler(Form1_LabelsChanged);
}

void Form1_LabelsChanged(CollectionExEventArgs evtArgs)
{
    MessageBox.Show("LabelsChanged event is fired" +
evtArgs.ChangeType.ToString() + evtArgs.Owner.ToString());
}
```

[VB]

```
Public Sub Form1_Load(ByVal sender As Object, ByVal e As EventArgs)
    AddHandler DirectCast(model1.EventSink,
DocumentEventSink).LabelsChanged, AddressOf Form1_LabelsChanged
End Sub
```

```
Private Sub Form1_LabelsChanged(ByVal evtArgs As CollectionExEventArgs)
    MessageBox.Show(("LabelsChanged event is fired" &
evtArgs.ChangeType.ToString()) + evtArgs.Owner.ToString())
End Sub
```

Layers Events

Programmatically, the events are written as follows:

[C#]

```
public void Form1_Load(object sender, EventArgs e)
{
    ((DocumentEventSink)model1.EventSink).LayersChanged += new
CollectionExEventHandler(Form1_LayersChanged);
    Layer layer0 = new Layer();
    this.diagram1.Model.Layers.Add(layer0);
    layer0.Enabled = true;
    layer0.Visible = true;
}

void Form1_LayersChanged(CollectionExEventArgs evtArgs)
{
    MessageBox.Show("LayersChanged event is fired." + "\n" + "Owner: "
+ evtArgs.Owner.ToString());
}
```

[VB]

```
Public Sub Form1_Load(ByVal sender As Object, ByVal e As EventArgs)
    AddHandler DirectCast(model1.EventSink,
DocumentEventSink).LayersChanged, AddressOf Form1_LayersChanged
    Dim layer0 As New Layer()
    Me.diagram1.Model.Layers.Add(layer0)
    layer0.Enabled = True
    layer0.Visible = True
End Sub

Private Sub Form1_LayersChanged(ByVal evtArgs As CollectionExEventArgs)
    MessageBox.Show(("LayersChanged event is fired." & vbLf & "Owner:
") + evtArgs.Owner.ToString())
End Sub
```

Sample diagram is as follows,

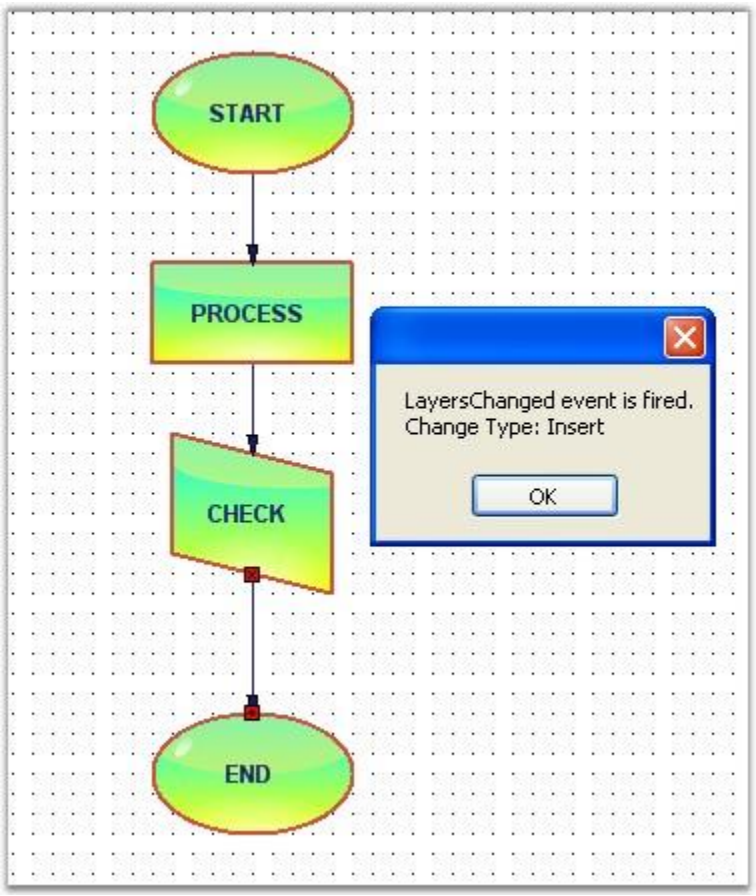

Figure 148: LayersChanged Event

4.6.9 Built-In Context Menu

Essential Diagram for Windows Forms provides Built-in Context Menu support for Diagram.

All available tools for Diagram control, File options, Edit options, Action options, Layout, Connectors and Shapes will be listed in the Built-in Context Menu.

Use Case Scenarios

This feature enables easy access of frequently used options.

Properties

Table 3: Property Table

Property	Description	Type	Data Type	Reference links
DefaultContextMenuEnabled	Used to enable default context menu.	NA	Boolean	NA

Enabling Default Context Menu

You can enable the default context menu using the *DefaultContextMenuEnabled* property.

The following code illustrates how to enable the default context menu:

[C#]

```
//show default context menu
diagram1.DefaultContextMenuEnabled = true;
```

[VB]

```
'show default context menu
diagram1.DefaultContextMenuEnabled = True
```

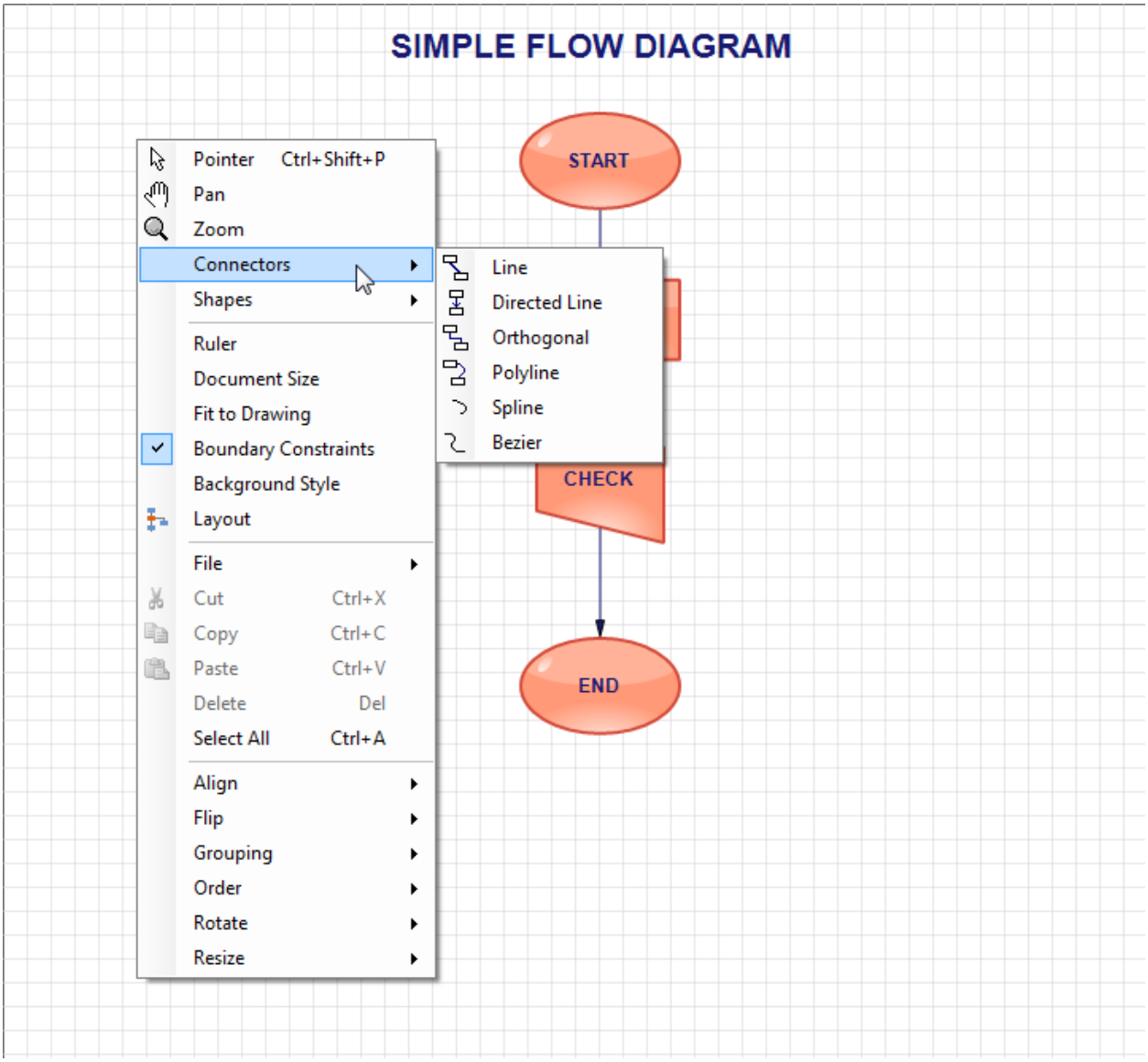

Figure 149: Default Context Menu

The following code illustrates how to disable the default context menu:

```
[C#]

//hide default context menu
            diagram1.DefaultContextMenuEnabled = false;
```



```
[VB]


'hide default context menu
            diagram1.DefaultContextMenuEnabled = False
```

Sample Link
To view a sample:

9. Open the Syncfusion Dashboard.
10. Click the **Windows Forms** drop-down list and select **Run Locally Installed Samples.**
11. Navigate to **Diagram Samples > Product Showcase > Diagram Builder.**

4.6.10 Adding Shapes by Clicking the Diagram Page

Essential Diagram enables you to draw the selected node by clicking the Diagram page instead of dragging from the Symbol Palette.

Properties

Table 4: Property Table

Property	Description	Type	Data Type	Reference links
Diagram	Reference to enable drawing the selected node by clicking on the diagram page.	NA	Diagram	NA.

Enabling Adding Shapes by Clicking Support
You can enable drawing shapes by clicking the diagram page using the *Diagram* property.

[C#]

```
            //Palette group view
```

```
paletteGroupView1.Diagram = diagram1;
        // Platte group bar
paletteGroupBar1.Diagram = diagram1;
```

[VB]

```
        'Palette group view
paletteGroupView1.Diagram = diagram1;
        'Platte group bar
paletteGroupBar1.Diagram = diagram1;
```

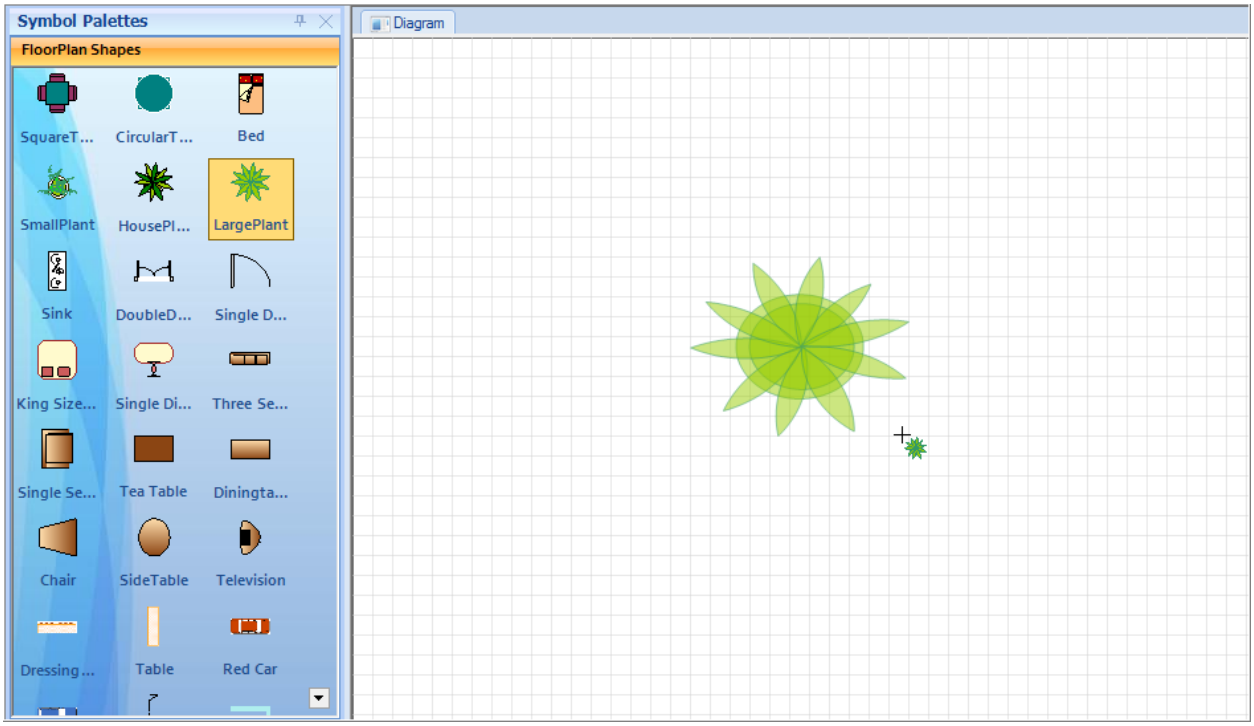

Figure 150: Added Shares

Note: Click the Diagram page to add the selected node. Click and drag to get the required size.

Sample Link

To view a sample:

1. Open the Syncfusion Dashboard.
2. Click the **Windows Forms** drop-down list and select **Run Locally Installed Samples.**
3. Navigate to **Diagram Samples > Product Showcase > Diagram Builder.**

4.6.11 Preview for Symbol Palette Item

Essential Diagram for Windows Forms provides preview support for Symbol Palette. When you drag an item from Symbol Palette to Diagram View, Preview of the dragged item will be displayed. You can enable or disable the preview support.

Use Case Scenario

This feature displays a preview of the item you drag from Symbol Palette, thus enables you to identify the item you are dragging from the symbol palette to Diagram view.

Properties

Table 5: Property Table

Property	Description	Type	Data Type	Reference links
ShowDragNodeCue	Gets or sets a value indicating whether preview is visible. The default value is true.	NA	Boolean	NA
DragNodeCueEnabled	Gets or sets a value indicating whether preview is enabled. The default value is true.	NA	Boolean	NA

Enabling Preview Support

To enable preview for the dragged item from Symbol Palette, set the *DragNodeCueEnabled* property of *PalatteGroupBar/PaletteGroupView* to true. To disable preview set this to false. By default this is set to true.

Following code example illustrates how to enable preview support:

[C#]

```
                //enable dragged node cue
            paletteGroupBar1.DragNodeCueEnabled = true;

                        paletteGroupView1.DragNodeCueEnabled = true;

                //show dragged node cue
            paletteGroupBar1.ShowDragNodeCue = true;

            paletteGroupView1.ShowDragNodeCue = true;
```

[VB]

```
              //enable dragged node cue
            paletteGroupBar1.DragNodeCueEnabled = True;
                        paletteGroupView1.DragNodeCueEnabled = True

              //show dragged node cue
            paletteGroupBar1.ShowDragNodeCue = True;
            paletteGroupView1.ShowDragNodeCue = True;
```

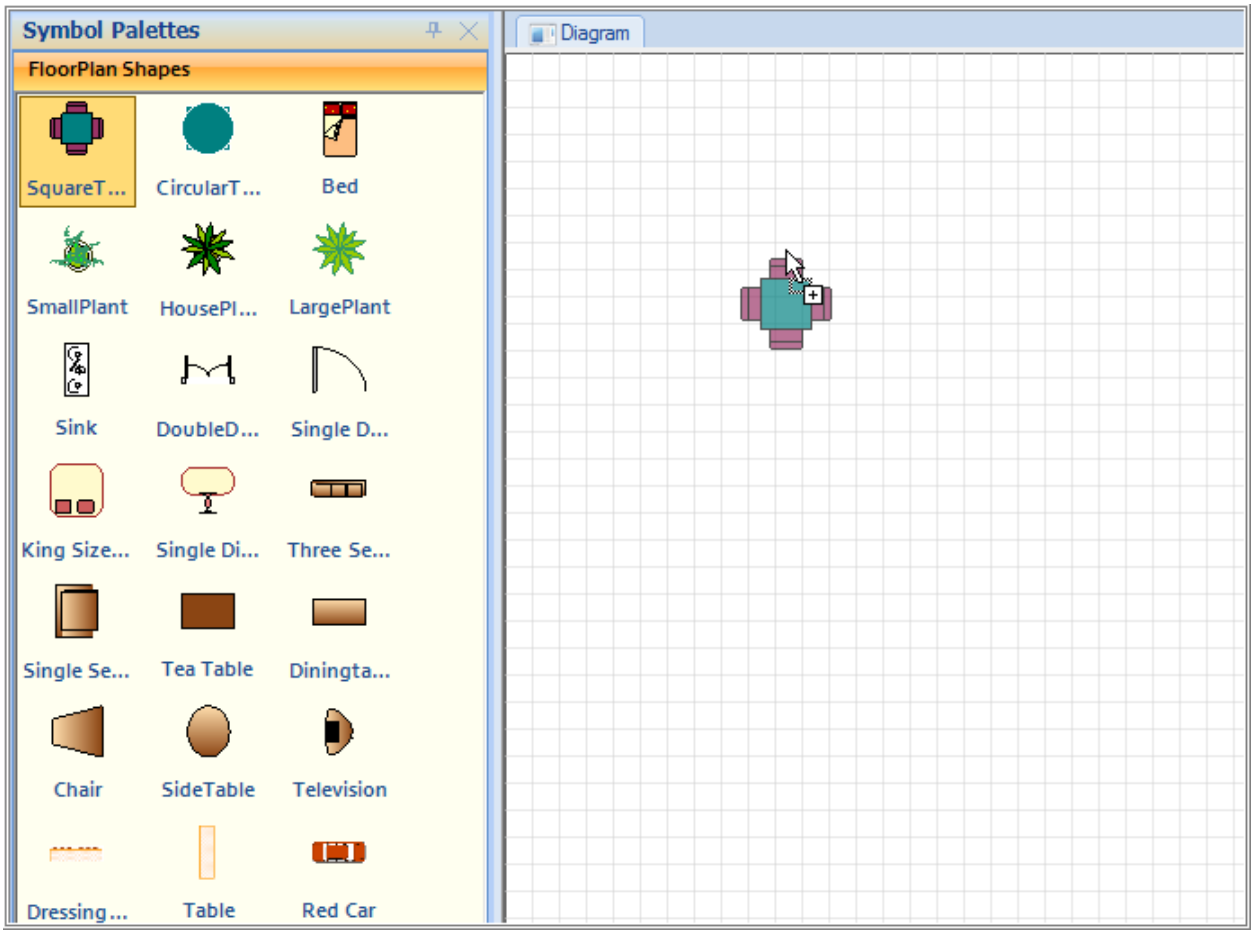

Figure 151: Preview of Dragged Item

The following code illustrates how to disable preview support:

```
[C#]
      //hide dragged node cue
            paletteGroupBar1.ShowDragNodeCue = false;
            paletteGroupView1.ShowDragNodeCue = false;
```

```
[VB]
              //hide dragged node cue
            paletteGroupBar1.ShowDragNodeCue = False;
            paletteGroupView1.ShowDragNodeCue = False;
```

Sample Link

To view a sample:

1. Open the Syncfusion Dashboard.
2. Click the **Windows Forms** drop-down list and select **Run Locally Installed Samples.**
3. Navigate to **Diagram Samples > Product Showcase > Diagram Builder.**

4.6.12 Dragging, Resizing, and Rotation Styles for Nodes

Essential Diagram for Windows Forms provides dragging, resizing, and rotation styles such as ghost copy, filled rectangle, solid outline, and dashed outline for nodes. These styles provide better visual effects for your diagram and increase the performance speed of the diagram while dragging, rotating, or resizing nodes.

Properties

Table 6: Properties Table

Property	Description	Type	Data Type
ResizingStyle	Gets or sets resizing style for the rendering helper	NA	RenderingHelperStyle
DraggingStyle	Gets or sets dragging style for the rendering helper	NA	RenderingHelperStyle
RotatingStyle	Gets or sets rotating style for the rendering helper	NA	RenderingHelperStyle

Applying Styles to Rendering Helper

The following code example illustrates how to apply styles to the rendering helper while resizing, dragging, and rotating nodes.

```
[C#]

  //Specify dragging, resizing, and rotation styles to the rendering helper

            diagram1.Controller.DraggingStyle =
RenderingHelperStyle.SolidOutline;

            diagram1.Controller.ResizingStyle =
RenderingHelperStyle.GhostCopy;

            diagram1.Controller.RotatingStyle =
RenderingHelperStyle.DashedOutline;
```

[VB]

```
  'Specify dragging, resizing, and rotation styles to the rendering helper
                  diagram1.Controller.DraggingStyle =
RenderingHelperStyle.SolidOutline
                  diagram1.Controller.ResizingStyle =
RenderingHelperStyle.GhostCopy
                  diagram1.Controller.RotatingStyle =
RenderingHelperStyle.DashedOutline
```


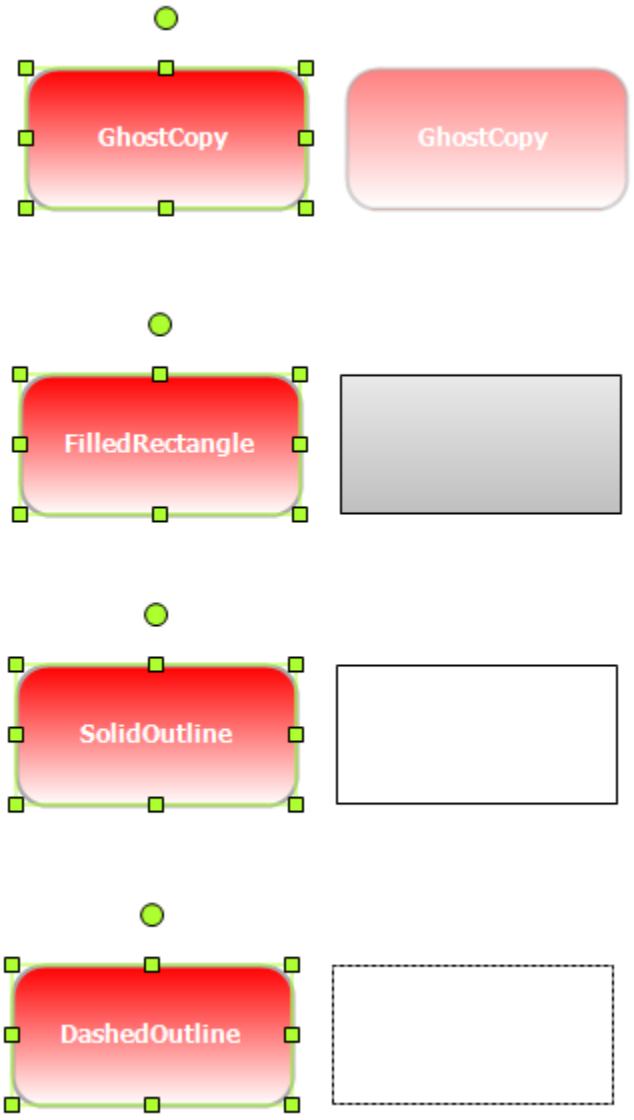

Figure 152: Diagram nodes with different styles

4.6.13 Dynamic Properties

This feature lets the user add additional properties or data to the nodes and connectors. Any type of data can be added as additional data or properties.

The node's PropertyBag property, which is a key value pair, is used to add, edit, and remove the additional properties or data and can be serialized when saving the diagram.

The diagram has built-in UI dialogs to add, edit, and remove the dynamic properties.

Use Case Scenario:
It is used to store additional data to the nodes or connectors as needed.

Properties

Property	Description	Data Type
PropertyBag	Gets or sets the dynamic property data dictionary.	Dictionary<string, object>

The following code shows how to add additional data to a node by using the PropertyBag property:

```
[C#]
node.PropertyBag.Add("Name", emply.EmployeeName);
node.PropertyBag.Add("ID", emply.EmployeeID);
node.PropertyBag.Add("Designation", emply.Designation);
```

```
[VB]
node.PropertyBag.Add("Name", emply.EmployeeName)
node.PropertyBag.Add("ID", emply.EmployeeID)
node.PropertyBag.Add("Designation", emply.Designation)
```

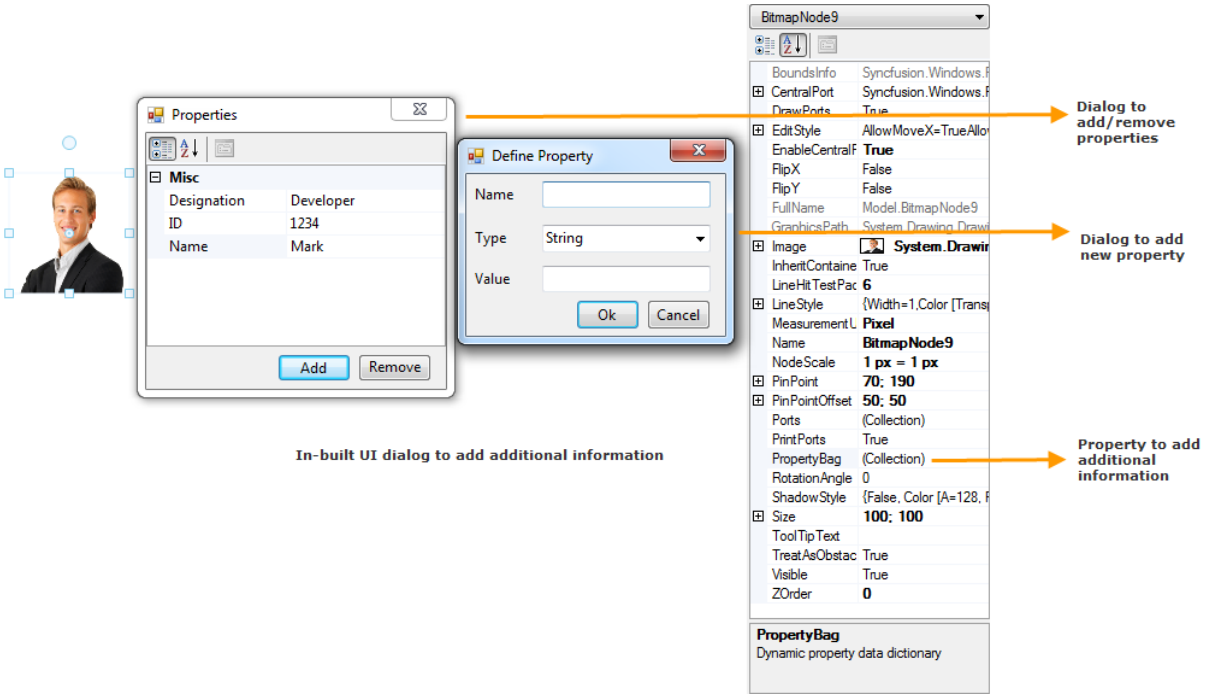

Figure 153: Dynamic Properties in a Diagram

5 Frequently Asked Questions

This section guides you through questions based on very specific tasks in Diagram control.

5.1 How To Add a Custom Property To Diagram And Display It In the Property Editor

The below given code snippet illustrates how you can add a custom property to Diagram and display it in the Property Editor.

```
[C#]

//Derived Diagram
public class MyDiagram :
Syncfusion.Windows.Forms.Diagram.Controls.Diagram
{
    public override Model CreateModel()
    {
        return new QuickStart.MainForm.MyModel();
    }
}

//Derived Model where the new property is added
public class MyModel : Syncfusion.Windows.Forms.Diagram.Model
{
    public override void SetDefaultPropertyValues()
    {
        base.SetDefaultPropertyValues();
        this.SetPropertyValue("MyCustomProperty", 0);
    }
    [
    Browsable(true),
    Category("MyProperties"),
    Description("Description for MyCustomProperty")]
    public int MyCustomProperty
    {
        get
        {
            object value = this.GetPropertyValue("MyCustomProperty");
            if (value != null)
            {
                return (int)value;
            }
```

```
            return 0;
        }
    }
}
```

```
[VB.NET]

'Derived Diagram
Public Class MyDiagram
Inherits Syncfusion.Windows.Forms.Diagram.Controls.Diagram
   Public Overrides Function CreateModel() As Model
      Return New QuickStart.MainForm.MyModel()
   End Function 'CreateModel
End Class

'Derived Model where the new property is added
Public Class MyModel
   Inherits Syncfusion.Windows.Forms.Diagram.Model
   Public Overrides Sub SetDefaultPropertyValues()
      MyBase.SetDefaultPropertyValues()
      Me.SetPropertyValue("MyCustomProperty", 0)
   End Sub 'SetDefaultPropertyValues
   Public ReadOnly Property MyCustomProperty() As Integer
      Get
      Dim value As Object = Me.GetPropertyValue("MyCustomProperty")
      If Not (value Is Nothing) Then
         Return Fix(value)
      End If
         Return 0
      End Get
   End Property
End Class 'MyModel
```

5.2 How To Add Ports To A Custom Symbol

The following code snippet illustrates how ports can be added to a custom symbol.

```
[C#]

private CirclePort leftport;
private CirclePort rightport;
```



```
//Add these lines to MySymbol's Constructor
//Port locations
leftport = new CirclePort(new PointF(0, this.Height / 2));
rightport = new CirclePort(new PointF(this.Width, this.Height / 2));

//Append CirlePorts to MySymbol
AppendChild(leftport);
AppendChild(rightport);

//Make CenterPort visible
this.CenterPort.Visible = true;
```

```
[VB.NET]

Private leftport As CirclePort
Private rightport As CirclePort

'Add these lines to MySymbol's Constructor
'Port locations
leftport = New CirclePort(New PointF(0, Me.Height / 2))
rightport = New CirclePort(New PointF(Me.Width, Me.Height / 2))

'Append CirlePorts to MySymbol
AppendChild(leftport)
AppendChild(rightport)

'Make CenterPort visible
Me.CenterPort.Visible = True
```

5.3 How To Change the Color Of the LineConnector When Activating the LineTool

We can change the color of the LineConnector while activating the LineTool using the **LineTool** and **LineBase** class. In the Mouse up event of the LineTool class, change the color of the link.

Refer to the following code snippet in CustomLineConnector class.

```
[C#]
```



```
public override Tool ProcessMouseUp( MouseEventArgs evtArgs )
{
    CompleteAction( ptStart, ptEnd );
}

private void CompleteAction( PointF ptStart, PointF ptEnd )
{
    Node node = CreateNode( ptStart, ptEnd );
    node.LineStyle.LineColor = Color.Red;
}

// To activate the LineTool in MainForm.cs.

MyLineTool linetool = new MyLineTool(this.diagram1.Controller);
this.diagram1.Controller.RegisterTool(linetool);
this.diagram1.Controller.ActivateTool(linetool);
```

[VB.NET]

```
Public Overloads Function ProcessMouseUp(ByVal evtArgs As
MouseEventArgs) As Syncfusion.Windows.Forms.Diagram.Tool
   CompleteAction(ptStart, ptEnd)
End Function

Private Sub CompleteAction(ByVal ptStart As PointF, ByVal ptEnd As
PointF)
   Dim node As Syncfusion.Windows.Forms.Diagram.Node =
   CreateNode(ptStart, ptEnd)
   node.LineStyle.LineColor = Color.Red
End Sub

' To activate the LineTool in MainForm.cs.
Dim linetool As New MyLineTool(Me.diagram1.Controller)
Me.diagram1.Controller.RegisterTool(linetool)
Me.diagram1.Controller.ActivateTool(linetool)
```


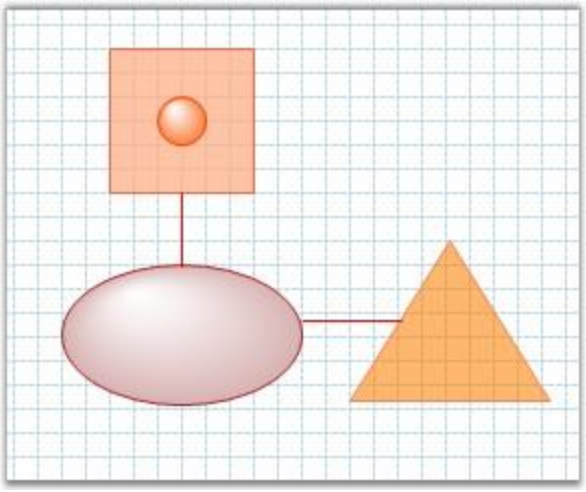

Figure 154: LineColor = "Red"

5.4 How to Change the Selection Mode of the SelectTool

The Diagram SelectTool provides an enum property called *SelectMode*, to change the selection mode. The following are the supported selection modes:

- **Containing** - Specific objects that are fully enveloped by the tracking rectangle will be selected by the tool.
- **Intersecting** - Specific objects that are intersecting the tracking rectangle will be selected by the tool.

Containing is the default selection mode.

The following code snippet illustrates how to change the selection mode at runtime:

```
[C#]


((DiagramViewerEventSink)diagram1.EventSink).ToolActivated += new
ToolEventHandler(Diagram_ToolActivated);

private void Diagram_ToolActivated(ToolEventArgs evtArgs)

        {

            if (evtArgs.Tool is SelectTool)

            {

                //change the SelectionMode as "Intersecting" which
Specifies that objects intersecting the tracking rectangle will be
selected by the tool.
```

```
                ((SelectTool)evtArgs.Tool).SelectMode =
SelectMode.Intersecting;
            }
        }
```

```
[VB]


AddHandler (CType(Diagram1.EventSink,
DiagramViewerEventSink)).ToolActivated, AddressOf Diagram_ToolActivated
Private Sub Diagram_ToolActivated(ByVal evtArgs As ToolEventArgs)
        If TypeOf evtArgs.Tool Is SelectTool Then
            'change the SelectionMode as "Intersecting" which Specifies
that objects intersecting the tracking rectangle will be selected by
the tool.
            CType(evtArgs.Tool, SelectTool).SelectMode =
SelectMode.Intersecting
        End If
    End Sub
```

5.5 How To Combine Different Actions Into One Atomic Action To Avoid the Undo Operation On Certain Actions

This is done by calling the Model.HistoryManger.StartAtomicAction(string description) / EndAtomicAction() methods.

The actions can be recorded into the history manager such that the undo and redo operations can be performed. The recording can be controlled and the undo and redo actions can be performed using the following tools.

- **StartAtomicAction**-This tool, when called, stops recording the actions and hence will not be added to the undo history manager.
- **EndAtomicAction**-This tool cancels the StartAtomicAction process and turns on the recording of actions in the history manager.
-

```
[C#]
```



```
this.diagram1.Model.HistoryManager.StartAtomicAction("Custom Action");
this.diagram1.Model.HistoryManager.EndAtomicAction();
```

```
[VB.NET]

Me.diagram1.Model.HistoryManager.StartAtomicAction("Custom Action")
Me.diagram1.Model.HistoryManager.EndAtomicAction()
```

5.6 How To Control the Number Of Connections That Can Be Drawn From / To the Port

This can be done using the port's **ConnectionsLimit** property. ConnectionsLimit specifies the number of connections to be allowed. Default value is ***10***.

```
[C#]

Syncfusion.Windows.Forms.Diagram.ConnectionPoint cp = new
Syncfusion.Windows.Forms.Diagram.ConnectionPoint();
cp.ConnectionsLimit = 12;
```

```
[VB.NET]

Dim cp As New Syncfusion.Windows.Forms.Diagram.ConnectionPoint()
cp.ConnectionsLimit = 12
```

5.7 How To Convert Diagram Node To Any Image

To export a single node as an image, use the below code snippet.

```
[C#]

Node selectedNode = this.diagram1.Controller.SelectionList[0];
if (selectedNode != null)
{
    //Node size depends on the border line. So we have to calculate the
Line width also.
    Bitmap bmp = new Bitmap(Convert.ToInt32(selectedNode.Size.Width +
```



```
selectedNode.LineStyle.LineWidth),
    Convert.ToInt32(selectedNode.Size.Height +
selectedNode.LineStyle.LineWidth));

    using (Graphics gr = Graphics.FromImage(bmp))
    {

        System.Drawing.Drawing2D.Matrix m = new
System.Drawing.Drawing2D.Matrix();
        m.Translate(Convert.ToInt32(-
selectedNode.BoundingRectangle.Left), Convert.ToInt32(-
selectedNode.BoundingRectangle.Top));
        gr.Transform = m;

        //This will render the node image in a graphics surface and we
can export it easily to any type of image.
        selectedNode.Draw(gr);
        bmp.Save("image.png", System.Drawing.Imaging.ImageFormat.Png);
        Process.Start("image.png");
    }
}
```

[VB.NET]

```
Dim selectedNode As Syncfusion.Windows.Forms.Diagram.Node =
Me.Diagram1.Controller.SelectionList(0)
   If selectedNode IsNot Nothing Then
   'Node size depends on the border line. So we have to calculate the
   Line width also.
   Dim bmp As New Bitmap(Convert.ToInt32(selectedNode.Size.Width +
   selectedNode.LineStyle.LineWidth),
   Convert.ToInt32(selectedNode.Size.Height +
   selectedNode.LineStyle.LineWidth))
   Using gr As Graphics = Graphics.FromImage(bmp)
      Dim m As New System.Drawing.Drawing2D.Matrix()
      m.Translate(Convert.ToInt32(-
      selectedNode.BoundingRectangle.Left), Convert.ToInt32(-
      selectedNode.BoundingRectangle.Top))
      gr.Transform = m

      'This will render the node image in a graphics surface and we can
      export it easily to any type of image.
      selectedNode.Draw(gr)
      bmp.Save("image.png", System.Drawing.Imaging.ImageFormat.Png)
      Process.Start("image.png")
   End Using
End If
```


5.8 How To Copy / Paste Nodes In Essential Diagram

The following code snippet illustrates how you can copy / paste nodes (symbol, shape, or link) in Essential Diagram.

[C#]

```
//Copy Code
this.diagram1.Controller.Copy();

//Paste Code
//If the data in the clipboard is of the type ClipboardNodeCollection,
paste it onto the Diagram.
IDataObject clipboardData = Clipboard.GetDataObject();
if (clipboardData.GetDataPresent(typeof(ClipboardNodeCollection)))
{
   this.diagram1.Controller.Paste();
}
```

[VB.NET]

```
'Copy Code
Me.diagram1.Controller.Copy()

'Paste Code
'If the data in the clipboard is of the type ClipboardNodeCollection,
paste it onto the Diagram.
Dim clipboardData As IDataObject = Clipboard.GetDataObject()
If clipboardData.GetDataPresent(Type.GetType(ClipboardNodeCollection))
Then
   Me.diagram1.Controller.Paste()
End If
```

5.9 How To Create a Connection Programmatically

Typically, symbols and links are connected together using the interactive **LinkTool** UI tool or the **LinkCmd** command class. Sometimes, it is useful to create connections programmatically. For example, you might be generating a diagram from the data in a database or possibly even writing your own custom link tool. You might even create a connection between two symbols directly without having a link in between.

The following code creates a link and connects it to the center ports of two symbols.

```
[C#]

public Link LinkSymbols(Symbol sym1, Symbol sym2)
{
    Link link = new Link(Link.Shapes.Line);
    sym1.Connect(sym1.CenterPort, link.TailPort);
    sym2.Connect(link.HeadPort, sym2.CenterPort);
    return link;
}
```

```
[VB.NET]

Public Function LinkSymbols(ByVal sym1 As
Syncfusion.Windows.Forms.Diagram.Symbol, ByVal sym2 As
Syncfusion.Windows.Forms.Diagram.Symbol) As LinkLabel.Link
   Dim link As LinkLabel.Link = New LinkLabel.Link(link.Shapes.Line)
   sym1.Connect(sym1.CenterPort, link.TailPort)
   sym2.Connect(link.HeadPort, sym2.CenterPort)
   Return link
End Function
```

5.10 How To Create a Custom Symbol

The following code sample demonstrates how you can create a custom symbol and use it in Essential Diagram.

1. Create the custom symbol.

```
[C#]

// Custom Symbol (MySymbol.cs)
public class MySymbol : Symbol
{
    private Syncfusion.Windows.Forms.Diagram.Rectangle outerRect =
```



```
null;
    private Ellipse innerEllipse = null;

    public MySymbol()
    {
       // Add child nodes to the symbol programmatically.
       // Add an outer rectangle.
        this.outerRect = new
Syncfusion.Windows.Forms.Diagram.Rectangle(0, 0, 120, 80);
        this.outerRect.Name = "Rectangle";
        this.outerRect.FillStyle.Color = Color.Khaki;
        this.AppendChild(outerRect);

        // Add an inner ellipse.
        this.innerEllipse = new Ellipse(10, 10, 100, 60);
        this.innerEllipse.Name = "Ellipse";
        this.AppendChild(innerEllipse);

        // Add Label.
        Label lbl = this.AddLabel("My Symbol", BoxPosition.Center);
        lbl.BackgroundStyle.Color = Color.Transparent;
    }
}
```

```
[VB.NET]

' Custom Symbol (MySymbol.vb)
Public Class MySymbol
        Inherits Symbol
        Private outerRect As Syncfusion.Windows.Forms.Diagram.Rectangle
= Nothing
        Private innerEllipse As
Syncfusion.Windows.Forms.Diagram.Ellipse = Nothing

        Public Sub New()

            ' Add child nodes to the symbol programmatically.

            ' Add an outer rectangle.
            Me.outerRect = New
Syncfusion.Windows.Forms.Diagram.Rectangle(0, 0, 120, 80)
            Me.outerRect.Name = "Rectangle"
            Me.outerRect.FillStyle.Color = Color.Khaki
            Me.AppendChild(outerRect)

            ' Add an inner ellipse.
```



```
            Me.innerEllipse = New
Syncfusion.Windows.Forms.Diagram.Ellipse(10, 10, 100, 60)
            Me.innerEllipse.Name = "Ellipse"
            Me.AppendChild(innerEllipse)

            ' Add Label.
            Dim lbl As Label = Me.AddLabel("My Symbol",
BoxPosition.Center)
            lbl.BackgroundStyle.Color = Color.Transparent
        End Sub ' New
End Class ' MySymbol
```

2. Use the symbol in the form.

```
[C#]

// Register InsertTool for MySymbol.
this.diagram1.Controller.RegisterTool(new
InsertSymbolTool("InsertMySymbol", typeof(MySymbol)));

// Activate InsertTool for MySymbol.
this.diagram1.ActivateTool("InsertMySymbol");
```

```
[VB.NET]

' Register InsertTool for MySymbol.
Me.diagram1.Controller.RegisterTool(New
InsertSymbolTool("InsertMySymbol", GetType(MySymbol)))

' Activate InsertTool for MySymbol.
Me.diagram1.ActivateTool("InsertMySymbol")
```

5.11 How To Create a Directional Link

Links can be provided with end point decorators to convey the direction. The following code snippet shows how to create a directional link by adding a 'Filled Arrow' end point visual to the head port edge of the link.

```
[C#]

// Create a directional link.
Link link = new Link(pts);
```



```
EndPointDecoratorModel decoratorMdl =
Global.EndPointDecoratorPalette["Filled Arrow"];
if (decoratorMdl != null)
{
 link.EndPoints.LastEndPointDecorator = decoratorMdl.CreateInstance();
}
```

```
[VB.NET]

' Create a directional link.
Dim link As New LinkLabel.Link(pts)
Dim decoratorMdl As EndPointDecoratorModel =
Global.EndPointDecoratorPalette("Filled Arrow")
If Not (decoratorMdl Is Nothing) Then
 link.EndPoints.LastEndPointDecorator = decoratorMdl.CreateInstance()
End If
```

5.12 How To Detect Whether a New Link Has Been Added / Removed From a Diagram

You can use the **Diagram.Model.ConnectionsChangeComplete** event to detect whether a new link has been added / removed from a diagram.

The following code snippet illustrates how to detect a new link that has been added to the diagram, and the symbols connected by the new link. It also illustrates how to remove a link that connects symbols of the same type.

```
[C#]

// Use the Diagram.ConnectionsChangeComplete event to be notified of
the creation of a new Link.
// The Link.FromNode and Link.ToNode properties provide access to the
symbols that the link connects.
private void diagram1_ConnectionsChangeComplete(object sender,
Syncfusion.Windows.Forms.Diagram.ConnectionCollectionEventArgs evtArgs)
{
    if ((evtArgs.ChangeType ==
Syncfusion.Windows.Forms.Diagram.CollectionExChangeType.Insert) &&
(evtArgs.Connection != null))
    {
        Connection newconn = evtArgs.Connection;
```

```
        Link newlink = null;
        if (newconn.SourcePort is LinkPort)
            newlink = newconn.SourcePort.Container as Link;
        else if (newconn.TargetPort is LinkPort)
            newlink = newconn.TargetPort.Container as Link;
        if ((newlink != null) && (newlink.FromNode != null) &&
(newlink.ToNode != null))
        {
            Trace.WriteLine("A new link was added to the Diagram");
            Symbol tailsymbol = newlink.FromNode as Symbol;
            Symbol headsymbol = newlink.ToNode as Symbol;
            if ((tailsymbol.Nodes.Count == headsymbol.Nodes.Count) &&
(tailsymbol.Nodes[0].Name == headsymbol.Nodes[0].Name))
            {
                // Comparing the symbol's child nodes count and child
node type is a simplistic way
                // to determine whether the two symbols are of the same
type.
                Trace.WriteLine("The two symbols are of the same
type.");

                // Use the RemoveNodesCmd to delete the new link
                RemoveNodesCmd removecmd = new RemoveNodesCmd();
                removecmd.Nodes.Add(newlink);
                this.diagram1.Controller.ExecuteCommand(removecmd);
            }
        }
    }
}
```

[VB.NET]

```
Private Sub diagram1_ConnectionsChangeComplete(ByVal sender As Object,
ByVal evtArgs As
Syncfusion.Windows.Forms.Diagram.ConnectionCollectionEventArgs) Handles
diagram1.ConnectionsChangeComplete
    If evtArgs.ChangeType =
Syncfusion.Windows.Forms.Diagram.CollectionExChangeType.Insert AndAlso
Not (evtArgs.Connection Is Nothing) Then
        Dim newconn As Connection = evtArgs.Connection
        Dim newlink As Link = Nothing
        If TypeOf newconn.SourcePort Is LinkPort Then
            newlink = newconn.SourcePort.Container
        End If
        If TypeOf newconn.TargetPort Is LinkPort Then
            newlink = newconn.TargetPort.Container
        End If
```



```
        If Not (newlink Is Nothing) AndAlso Not (newlink.FromNode Is
Nothing) AndAlso Not (newlink.ToNode Is Nothing) Then
            Trace.WriteLine("A new link was added to the Diagram")
            Dim tailsymbol As Symbol = newlink.FromNode
            Dim headsymbol As Symbol = newlink.ToNode
            If tailsymbol.Nodes.Count = headsymbol.Nodes.Count AndAlso
tailsymbol.Nodes(0).Name = headsymbol.Nodes(0).Name Then
                ' Comparing the symbol's child nodes count and child
node type is a simplistic way
                ' to determine whether the two symbols are of the same
type.
                Trace.WriteLine("The two symbols are of the same
type.")

                ' Use the RemoveNodesCmd to delete the new link
                Dim removecmd As New RemoveNodesCmd
                removecmd.Nodes.Add(newlink)
                Me.diagram1.Controller.ExecuteCommand(removecmd)
            End If
        End If
    End If
End Sub
```

5.13 How To Detect Whether a New Symbol Or Shape Has Been Added / Removed From A Diagram

You can make use of the **Diagram.Model.EventSink.NodeCollectionChanged** to detect whether a new node (symbol, shape or link) has been added/removed from a diagram. The event's **CollectionExEventArgs** argument provides information about the node ensuing the add / remove operation.

The following code snippet updates a label with information on the type of the node that is added/deleted from the diagram.

```
[C#]

diagram1.Model.EventSink.NodeCollectionChanged += new
CollectionExEventHandler(EventSink_NodeCollectionChanged);

//ChildrenChangeComplete Event
//Update Label2 depending on whether a Shape is added or deleted from
the Diagram
```



```
 private void EventSink_NodeCollectionChanged(CollectionExEventArgs
evtArgs)
 {
            Node n = evtArgs.Element as Node;

            //ChangeType specifies whether the Collection change is
Insertion/Removal
            if (evtArgs.ChangeType == CollectionExChangeType.Insert)
            {
                this.label2.ForeColor = Color.Blue;

                //Gets the Name of the inserted element
                this.label2.Text = "Last Node Added : " +
n.Name.ToString();
            }
            else if (evtArgs.ChangeType ==
CollectionExChangeType.Remove)
            {
                this.label2.ForeColor = Color.Red;

                //Gets the Name of the removed element
                this.label2.Text = "Last Node Removed : " +
n.Name.ToString();
            }
 }
```

[VB.NET]

```
EventSink.NodeCollectionChanged += new
CollectionExEventHandler(EventSink_NodeCollectionChanged)

Me.Diagram1.Model.EventSink.NodeCollectionChanged+=New
CollectionExEventHandler(EventSink_NodeCollectionChanged)

 'ChildrenChangeComplete Event
 'Update Label2 depending on whether a Shape is added or deleted from
the Diagram
 Private Sub Model_ChildrenChangeComplete(ByVal evtArgs As
CollectionExEventArgs)
        Dim n As Node = TryCast(evtArgs.Element, Node)
        'ChangeType specifies whether the Collection change is
Insertion/Removal
        If evtArgs.ChangeType = CollectionExChangeType.Insert Then
            Me.label2.ForeColor = Color.Blue

            'Gets the Name of the inserted element
            Me.label2.Text = "Last Node Added: " + n.Name.ToString()
```



```
        ElseIf evtArgs.ChangeType = CollectionExChangeType.Remove Then
            Me.label2.ForeColor = Color.Red

            'Gets the Name of the removed element
            Me.label2.Text = "Last Node Removed: " + n.Name.ToString()
        End If
    End Sub
```

5.14 How To Display ToolTips For the Symbols

ToolTips can be displayed using the Model's **MouseEnter** and **MouseLeave** events. Here is a sample where tooltips are displayed only for nodes of the type 'MySymbol'.

[C#]

```
private void Model_MouseEnter(object sender, NodeMouseEventArgs
evtArgs)
{
    if (evtArgs.Node.GetType() == typeof(MySymbol))
    {

        this.toolTip1.SetToolTip(this.diagram1,
evtArgs.Node.Name.ToString());

        this.toolTip1.Active = true;
    }
}

private void Model_MouseLeave(object sender, NodeMouseEventArgs
evtArgs)
{

    this.toolTip1.Active = false;
}
```

[VB.NET]

```
Private Sub Model_MouseEnter(ByVal sender As Object, ByVal evtArgs As
Syncfusion.Windows.Forms.Diagram.NodeMouseEventArgs)
   If evtArgs.Node.Name.StartsWith("MySymbol") Then
      Me.toolTip1.SetToolTip(Me.Diagram1, evtArgs.Node.Name.ToString())
      Me.toolTip1.Active = True
```

```
   End If
End Sub

Private Sub Model_MouseLeave(ByVal sender As Object, ByVal evtArgs As
Syncfusion.Windows.Forms.Diagram.NodeMouseEventArgs)
   Me.toolTip1.Active = False
End Sub
```

5.15 How To Draw Custom Handles For Nodes Using the CustomHandleRenderer Property

Syncfusion Diagram provides users with the facility to draw their own handles for nodes using the **CustomHandleRenderer** property which is available in the View class. This property accepts the instances of the **UserHandleRenderer** class which acts as a base class for the custom Handle Renderer. By using this class, we can derive a class to create our own Handle Renderer and assign it to the **View.CustomHandleRenderer** property. If this property is assigned as 'Null', it will render the default style of the Handle.

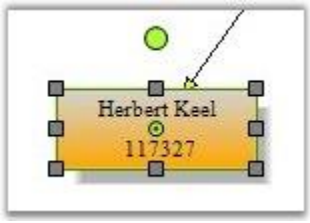

Figure 155: Default Appearance

UserHandleRenderer class methods are made as public virtual; this class duplicates the HandleRenderer functionality but has more convenient structure for overriding the default appearance, and it provides an option to implement the new handle renderer with minimal changes by deriving it. If you are having a source code license, you can see the complete source UserHandleRenderer in the following location for understanding more about handle renderers: Diagram.Base\Src\Utility\HandleRenderer.cs.

Here is a sample code snippet.

```
[C#]

public class MyHandleRenderer : UserHandleRenderer
{
```



```
    /// <summary>
    /// Draws endpoint outline and background.
    /// Override this method to change endpoint outline and background.
    /// </summary>
    /// <param name="gfx"></param>
    /// <param name="rectHandle"></param>
    /// <param name="endpoint"></param>
    public override void OnDrawEndPointOutline(Graphics gfx, RectangleF
rectHandle, EndPoint endpoint)
    {
        using (Pen pen = new Pen(Color.Black))
        {
            pen.Width = 1f / gfx.PageScale;
            pen.DashStyle = DashStyle.Dot;
            // Create brush to fill PinPoint interiors
            using (SolidBrush brush = new SolidBrush(Color.Yellow))
            {
                // if endpoint is connected with port -- fill its
interiors with red
                if (endpoint.Port != null)
                {
                    brush.Color = Color.Orange;
                }

                if (!endpoint.AllowMoveX && !endpoint.AllowMoveY)
                    brush.Color = Color.Gray;
                // Fill handle interiors
                gfx.FillRectangle(brush, rectHandle);

                // Outline handle
                gfx.DrawRectangle(pen,
                    rectHandle.X, rectHandle.Y, rectHandle.Width,
rectHandle.Height);
            }
        }
    }


    /// <summary>
    /// Method creates and returns GraphicsPath that represents
rotation handle
    /// Override this method to change rotationhandle appearance
    /// </summary>
    /// <param name="location">Location of rotation handle</param>
    /// <param name="handleSize">size of rotation handle</param>
    /// <returns></returns>
```



```
    public override GraphicsPath CreateRotationHandleShape(PointF
location, SizeF handleSize)
    {
        RectangleF rect = Geometry.CreateRect(location, handleSize);
        return PathFactory.CreateRoundRectangle(rect, 2);
    }


    /// <summary>
    /// Renders resize handle.
    /// Override this method to change appearance of resize handle.
    /// </summary>
    /// <param name="gfx">Target Graphics</param>
    /// <param name="size">size of outlining rectangle</param>
    public override void OnDrawResizeHandleShape(Graphics gfx,
BoxPosition handle, Node node, RectangleF rectHandle)
    {
        using (Pen pen = new Pen(m_handleOutlineColor))
        {
            pen.Width = 1f / gfx.PageScale;
            pen.DashStyle = DashStyle.Solid;
            pen.Color = HandleOutlineColor;

            // Create brush to fill PinPoint interiors
            using (Brush brush =
                (!Enabled(handle, node) ? new SolidBrush(Color.Red) :
                new SolidBrush(Color.Green)))
            {
                using (GraphicsPath gp =
PathFactory.CreateRoundRectangle(rectHandle, 3))
                {
                    gfx.FillPath(brush, gp);
                    // Outline handle
                    gfx.DrawPath(pen, gp);
                }
            }
        }
    }
}
```

[VB.NET]

```
Public Class MyHandleRenderer
Inherits UserHandleRenderer

        ''' <summary>
```

```
        ''' Draws endpoint outline and background.
        ''' Override this method to change endpoint outline and
background.
        ''' </summary>
        ''' <param name="gfx"></param>
        ''' <param name="rectHandle"></param>
        ''' <param name="endpoint"></param>
        Public Overloads Sub OnDrawEndPointOutline(ByVal gfx As
Graphics, ByVal rectHandle As RectangleF, ByVal endpoint As
Net.EndPoint)
            Using pen As New Pen(Color.Black)
                pen.Width = 1.0F / gfx.PageScale
                pen.DashStyle = DashStyle.Dot
                ' Create brush to fill PinPoint interiors
                Using brush As New SolidBrush(Color.Yellow)
                    ' if endpoint is connected with port -- fill its
interiors with red
                    If endpoint.Port IsNot Nothing Then
                        brush.Color = Color.Orange
                    End If

                    If Not endpoint.AllowMoveX AndAlso Not
endpoint.AllowMoveY Then
                        brush.Color = Color.Gray
                    End If
                    ' Fill handle interiors
                    gfx.FillRectangle(brush, rectHandle)

                    ' Outline handle
                    gfx.DrawRectangle(pen, rectHandle.X, rectHandle.Y,
rectHandle.Width, rectHandle.Height)
                End Using
            End Using
        End Sub


        ''' <summary>
        ''' Method creates and returns GraphicsPath that represents
rotation handle
        ''' Override this method to change rotationhandle appearance
        ''' </summary>
        ''' <param name="location">Location of rotation handle</param>
        ''' <param name="handleSize">size of rotation handle</param>
        ''' <returns></returns>
        Public Overloads Function CreateRotationHandleShape(ByVal
location As PointF, ByVal handleSize As SizeF) As
Drawing2D.GraphicsPath
```

```
            Dim rect As RectangleF = Geometry.CreateRect(location,
handleSize)
            Return PathFactory.CreateRoundRectangle(rect, 2)
        End Function


        ''' <summary>
        ''' Renders resize handle.
        ''' Override this method to change appearance of resize handle.
        ''' </summary>
        ''' <param name="gfx">Target Graphics</param>
        ''' <param name="size">size of outlining rectangle</param>
        Public Overloads Sub OnDrawResizeHandleShape(ByVal gfx As
Graphics, ByVal handle As Syncfusion.Windows.Forms.Diagram.BoxPosition,
ByVal node As Syncfusion.Windows.Forms.Diagram.Node, ByVal rectHandle
As RectangleF)
            Using pen As New Pen(m_handleOutlineColor)
                pen.Width = 1.0F / gfx.PageScale
                pen.DashStyle = DashStyle.Solid
                pen.Color = HandleOutlineColor

                ' Create brush to fill PinPoint interiors
                Using brush As Brush = (IIf(Not Enabled(handle, node),
New SolidBrush(Color.Red), New SolidBrush(Color.Green)))
                    Using gp As Drawing2D.GraphicsPath =
PathFactory.CreateRoundRectangle(rectHandle, 3)
                        gfx.FillPath(brush, gp)
                        ' Outline handle
                        gfx.DrawPath(pen, gp)
                    End Using
                End Using
            End Using
        End Sub
End Class
```

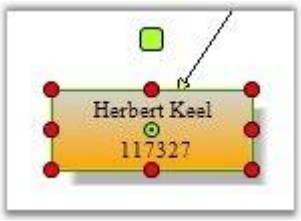

Figure 156: Customized Appearance

A sample which demonstrates this feature is available in the below sample installation location.

..\My Documents\Syncfusion\EssentialStudio\Version Number\Windows\Diagram.Windows\Samples\2.0\Getting Started\CustomSelectionHandle

5.16 How To Export a Diagram Into a Word Document

To export a diagram into a Word document, follow the below given steps.

1. Save the diagram in any one of the standard image formats such as bitmaps, enhanced metafiles, SVG format files, and so forth.

2. Export the saved images to the Word document using Essential DocIO.

 ***Note: To export the saved images to the Word document, you need to have Essential DocIO installed in your system.***

[C#]

```
System.Drawing.Image diagramimage = new Bitmap(1, 1,
PixelFormat.Format24bppRgb);
Graphics grfx = Graphics.FromImage(diagramimage);
IntPtr hdc = grfx.GetHdc();
Metafile emf = new Metafile(hdc, EmfType.EmfOnly);
Graphics emfgrfx = Graphics.FromImage(emf);
this.diagram1.View.ExportDiagramToGraphics(emfgrfx,true);
grfx.ReleaseHdc(hdc);
grfx.Dispose();
emfgrfx.Dispose();
diagramimage.Dispose();
WordDocument document = new WordDocument();

//Adding a new section to the document.
IWSection section = document.AddSection();

//Adding a paragraph to the section.
IWParagraph paragraph = section.AddParagraph();
WPicture mImage = (WPicture)paragraph.AppendPicture(emf);
document.Save("Sample.doc", Syncfusion.DocIO.FormatType.Doc);
System.Diagnostics.Process.Start("Sample.doc");
```

[VB.NET]


```
Dim diagramimage As System.Drawing.Image = New Bitmap(1, 1,
PixelFormat.Format24bppRgb)
Dim grfx As Graphics = Graphics.FromImage(diagramimage)
Dim hdc As IntPtr = grfx.GetHdc()
Dim emf As Metafile = New Metafile(hdc, EmfType.EmfOnly)
Dim emfgrfx As Graphics = Graphics.FromImage(emf)
Me.diagram1.View.ExportDiagramToGraphics(emfgrfx,True)
grfx.ReleaseHdc(hdc)
grfx.Dispose()
emfgrfx.Dispose()
diagramimage.Dispose()
Dim document As WordDocument = New WordDocument()

'Adding a new section to the document.
Dim section As IWSection = document.AddSection()

'Adding a paragraph to the section.
Dim paragraph As IWParagraph = section.AddParagraph()
Dim mImage As WPicture = CType(paragraph.AppendPicture(emf), WPicture)
document.Save("Sample.doc", Syncfusion.DocIO.FormatType.Doc)
System.Diagnostics.Process.Start("Sample.doc")
```

5.17 How To Generate a Thumbnail Image Of a Diagram

To display a thumbnail image of the diagram, follow the below given steps.

1. Generate a Bitmap image of the diagram.

2. Use the **GetThumbnailImage** method of the **Image** class to generate a thumbnail, and set it as the image of the picture box in which you want to display the thumbnail.

The following code illustrates this.

```
[C#]

public bool ThumbnailCallback()
{
   return false;
}

//Generate Thumbnail and set it to be image of the PictureBox
```



```
Image.GetThumbnailImageAbort myCallback = new
Image.GetThumbnailImageAbort(ThumbnailCallback);
this.pictureBox1.Image = (Bitmap)
diagramimage.GetThumbnailImage(150,75,myCallback, IntPtr.Zero);
```

```
[VB.NET]

Public Function ThumbnailCallback() As Boolean
   Return False
End Function

'Generate Thumbnail and set it to be image of the PictureBox
Dim myCallback As Image.GetThumbnailImageAbort = New
Image.GetThumbnailImageAbort(ThumbnailCallback)
Me.pictureBox1.Image =
CType(diagramimage.GetThumbnailImage(150,75,myCallback, IntPtr.Zero),
Bitmap)
```

5.18 How to Get a Connector Vertex Point?

Connector has a method called GetPoint to get its vertex point.

This method has two parameters: *int* and *bool*

- int specifies the vertex point in the local coordinates
- bool specifies the path of the connector

Set the bool parameter to True to get the connector vertex point based on its graphical path and False to get the connector point based on its relative path.

The following code snippet illustrates this:

```
[C#]


//LineConnector

ConnectorBase Connector = new LineConnector(new Drawing.PointF(0F, 0F),
new Drawing.PointF(10F, 10F));

//set points

Connector.SetPoints(new PointF[2] { Connector.GetPoint(0),
Connector.GetPoint(Connector.GetPoints().GetLength(0) - 1, false) });
```

```
[VB.NET]
```



```
'LineConnector
Dim Connector As ConnectorBase = New LineConnector(New
Drawing.PointF(0.0F, 0.0F), New Drawing.PointF(10.0F, 10.0F))
'set points
Connector.SetPoints(New PointF(1) {Connector.GetPoint(0),
Connector.GetPoint(Connector.GetPoints().GetLength(0) - 1, False)})
```

5.19 How to Get the Nearest Grid Point on a Diagram

The **GetNearestGridPoint method** can be used to get the nearest grid point on a diagram based on a given point.

This method has the following two parameters:

- **Point** - specifies the location which calculates the nearest grid point.
- **Int** - specifies the ruler height.

The following code snippet illustrates the implementation of **GetNearestGridPoint** method:

[C#]

```
//Current mouse position
Point ptMouse = new Point(e.X, e.Y);
int rulerHeight = (diagram1.ShowRulers) ? diagram1.RulersHeight : 0;
//Nearest grid point
PointF ptGridNearestPoint =
diagram1.View.Grid.GetNearestGridPoint(ptMouse, rulerHeight);
```

[VB.NET]

```
Current mouse position
Dim ptMouse As Point = New Point (e.X, e.Y)
Dim rulerHeight As Integer
rulerHeight = If((diagram1.ShowRulers), diagram1.RulersHeight, 0)
```

```
'Nearest grid point
Dim ptGridNearestPoint As PointF =
diagram1.View.Grid.GetNearestGridPoint(ptMouse, rulerHeight)
```

5.20 How To Hide Handles Completely From the Nodes

We can hide the handles completely by setting the **HandleColor** and **HandleOutlineColor** properties to 'Transparent' as follows.

```
[C#]

this.diagram1.View.HandleColor = Color.Transparent;
this.diagram1.View.HandleOutlineColor = Color.Transparent;
```

```
[VB.NET]

Me.diagram1.View.HandleColor = Color.Transparent
Me.diagram1.View.HandleOutlineColor = Color.Transparent
```

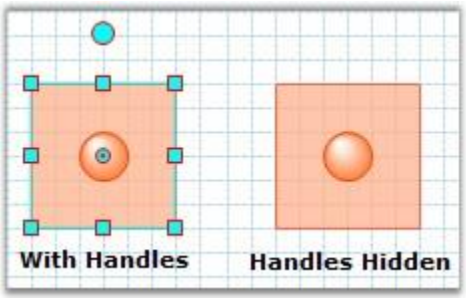

Figure 157: Illustrates Hiding Handles

5.21 How To Highlight a Particular Node At Run-time

A diagram node can be highlighted at run time using the mouse move actions. Using the Controller.GetNodeUnderMouseMove method, we can get the corresponding node under mouse move. By changing the node's LineStyle and FillStyle effects, we can highlight the respective nodes.

[C#]

```
Node globelNode;
Polygon PolygonNode;
public Timer timer1;
private void diagram1_MouseMove( object sender, MouseEventArgs e ) {
    try
    {
        // Retrieves node under the mouse action
        Node node1 = ( Node
)this.diagram1.Controller.GetNodeUnderMouse( new Point( e.X, e.Y ) );

        if ( node1 != null && ( node1.Name.ToString( ) != "TextNode" )
)
        {
            this.toolTip1.SetToolTip( this.diagram1, node1.Name );
            this.toolTip1.Active = true;

            globelNode = node1;
            PolygonNode = node1 as Polygon;
            defaultColor = PolygonNode.FillStyle.Color;

            this.timer1.Start( );
        }
        else
        {
            this.toolTip1.Active = false;
            this.timer1.Stop( );
        }
    }
    catch
    {
    }
}

private void timer1_Tick( object sender, EventArgs e ) {
    // Convert node as polygon
    Polygon poly = globelNode as Polygon;
    Random r = new Random( );

    if (poly != null)
    {
        //  Setting fillstyle for the ploygon
        poly.FillStyle.Color = Color.FromArgb(r.Next(255), r.Next(255),
r.Next(255));
        globelNode.LineStyle.DashStyle =
```



```
System.Drawing.Drawing2D.DashStyle.Dash;
        globelNode.LineStyle.LineWidth = 3;

        // Resetting the node with default values
        poly.FillStyle.Color = defaultColor;
        globelNode.LineStyle.DashStyle =
System.Drawing.Drawing2D.DashStyle.Solid;
        globelNode.LineStyle.LineWidth = 1;
    }
}
```

[VB.NET]

```
Private globelNode As Syncfusion.Windows.Forms.Diagram.Node
Private PolygonNode As Syncfusion.Windows.Forms.Diagram.Polygon
Public WithEvents timer1 As Timer

Private Sub diagram1_MouseMove(ByVal sender As Object, ByVal e As
MouseEventArgs) Handles Diagram1.MouseMove
   Try
            ' Retrieves node under the mouse action
            Dim node1 As Syncfusion.Windows.Forms.Diagram.Node =
CType(Me.Diagram1.Controller.GetNodeUnderMouse(New Point(e.X, e.Y)),
Syncfusion.Windows.Forms.Diagram.Node)

            If Not node1 Is Nothing AndAlso (node1.Name.ToString() <>
"TextNode") Then
                Me.toolTip1.SetToolTip(Me.diagram1, node1.Name)
                Me.toolTip1.Active = True

                globelNode = node1
                PolygonNode = CType(IIf(TypeOf node1 Is
Syncfusion.Windows.Forms.Diagram.Polygon, node1, Nothing),
Syncfusion.Windows.Forms.Diagram.Polygon)
                defaultColor = PolygonNode.FillStyle.Color

                Me.timer1.Start()
            Else
                Me.toolTip1.Active = False
                Me.timer1.Stop()
            End If
        Catch
   End Try
End Sub

Private Sub timer1_Tick(ByVal sender As Object, ByVal e As EventArgs)
Handles timer1.Tick
```



```
        ' Convert node as polygon
        Dim poly As Syncfusion.Windows.Forms.Diagram.Polygon =
CType(IIf(TypeOf globelNode Is
Syncfusion.Windows.Forms.Diagram.Polygon, globelNode, Nothing),
Syncfusion.Windows.Forms.Diagram.Polygon)
        Dim r As Random = New Random()
        If Not poly Is Nothing Then
            '  Setting fillstyle for the ploygon
            poly.FillStyle.Color = Color.FromArgb(r.Next(255),
r.Next(255), r.Next(255))
            globelNode.LineStyle.DashStyle =
System.Drawing.Drawing2D.DashStyle.Dash
            globelNode.LineStyle.LineWidth = 3

            ' Resetting the node with default values
            poly.FillStyle.Color = defaultColor
            globelNode.LineStyle.DashStyle =
System.Drawing.Drawing2D.DashStyle.Solid
            globelNode.LineStyle.LineWidth = 1
        End If
End Sub
```

A sample which demonstrates the node highlighting feature is available in the below sample installation location.

..\My Documents\Syncfusion\EssentialStudio\Version Number\Windows\Diagram.Windows\Samples\2.0\Getting Started\HighLightSample

5.22 How To Move / Place the Nodes Outside the Diagram Model's Bounds

Setting the model's **BoundaryConstraintsEnabled** property to 'False', will let you place the nodes outside the bounds of the diagram model.

[C#]

```
diagram1.Model.BoundaryConstraintsEnabled = false;
```

[VB.NET]

```
diagram1.Model.BoundaryConstraintsEnabled = False
```

5.23 How To Prevent the Nodes From Being Rotated

This can be done by raising the **Diagram.Model.EventSink.RotationChanging** event and cancelling the operation.

[C#]

```
this.diagram1.Model.EventSink.RotationChanging += new
RotationChangingEventHandler(EventSink_RotationChanging);

void EventSink_RotationChanging(RotationChangingEventArgs evtArgs)
{
    evtArgs.Cancel = true;
}
```

[VB.NET]

```
Me.diagram1.Model.EventSink.RotationChanging += New
RotationChangingEventHandler(EventSink_RotationChanging)

Private Sub EventSink_RotationChanging(ByVal evtArgs As
Syncfusion.Windows.Forms.Diagram.RotationChangingEventArgs)
   evtArgs.Cancel = True
End Sub
```

5.24 How To Print a Diagram In a Single Page

Essential Diagram uses the size of your diagram model and the printer page settings for calculating the number of pages to be rendered while printing. Even though you might have only one page worth of nodes in your diagram model, if the model bounds are larger, the Diagram control will attempt to paginate and print the entire model.

To print the diagram in a single page, you have to temporarily modify the size of the model.

[C#]

```
// The desired page size is 21 x 29.7 centimeters.
```

```
// The margins are of size 1 inch or 25 mm.
int verticalMargin = 260;
int horizontalMargin = 260;

// The following units are in millimeters, so convert them to pixels.
float pageHeight = Diagram.MeasureUnitsConverter.Convert((2970 -
(verticalMargin * 2)) / 10, MeasureUnits.Millimeter,
MeasureUnits.Pixel);

// float pageHeight = bounds.Height;
float pageWidth = Diagram.MeasureUnitsConverter.Convert((2100 -
(horizontalMargin * 2)) / 10, MeasureUnits.Millimeter,
MeasureUnits.Pixel);

// Set the model height to twice the page height.
diagram1.Model.DocumentSize.Height = (int)pageHeight / 2;

// Set the model width to page width
diagram1.Model.DocumentSize.Width = (int)pageWidth;
this.PrintPreview( );
```

[VB.NET]

```
' The desired page size is 21 x 29.7 centimeters.
' The margins are of size 1 inch or 25 mm.
Dim verticalMargin As Integer = 260
Dim horizontalMargin As Integer = 260

' The following units are in millimeters, so convert them to pixels.
Dim pageHeight As Single = Diagram.MeasureUnitsConverter.Convert((2970
- (verticalMargin * 2)) / 10, MeasureUnits.Millimeter,
MeasureUnits.Pixel)

' float pageHeight = bounds.Height;
Dim pageWidth As Single = Diagram.MeasureUnitsConverter.Convert((2100 -
(horizontalMargin * 2)) / 10, MeasureUnits.Millimeter,
MeasureUnits.Pixel)

' Set the model height to twice the page height.
diagram1.Model.Size = New SizeF(pageWidth, pageHeight / 2)
Me.PrintPreview( )
```

A sample which demonstrates Printing is available in the below sample installation path.

..\My Documents\Syncfusion\EssentialStudio\Version Number\Windows\Diagram.Windows\Samples\2.0\Getting Started\Printing

5.25 How To Programmatically Add a Symbol From the Palette

The following code sample demonstrates how you can programmatically add a symbol from the symbol palette to a diagram.

[C#]

```
if (paletteGroupView1.Palette.Nodes.Count > 0)
{
       Node nc =(Node)paletteGroupView1.Palette.Nodes[0].Clone();
       diagram1.Model.AppendChild(nc);
}
```

[VB.NET]

```
If paletteGroupView1.Palette.Nodes.Count > 0 Then
      Dim nc As Node =
DirectCast(paletteGroupView1.Palette.Nodes(0).Clone(), Node)
      diagram1.Model.AppendChild(nc)
End If
```

5.26 How To Programmatically Link Two Symbols

The **Syncfusion.Windows.Forms.Diagram.LinkCmd** command class can be used to programmatically link symbols.

The following code sample shows how to create a link between two symbols, symbol1 and symbol2.

[C#]

```
// Create a LinkCmd object.
LinkCmd linkcommand = new LinkCmd();
linkcommand.Link = new Link(Link.Shapes.Line);
```

```
// Set up the Source and Target ports for the Link.
linkcommand.SourcePort = symbol1.CenterPort;
linkcommand.TargetPort = symbol2.CenterPort;

// Execute the command to connect the two symbols.
this.diagram.Controller.ExecuteCommand(linkcommand);
```

```
[VB.NET]

'Create a LinkCmd object.
Dim linkcommand As New LinkCmd()
linkcommand.Link = New Link(Link.Shapes.Line)

'Set up the Source and Target ports for the Link.
linkcommand.SourcePort = symbol1.CenterPort
linkcommand.TargetPort = symbol2.CenterPort

'Execute the command to connect the two symbols.
Me.diagram.Controller.ExecuteCommand(linkcommand)
```

5.27 How To Retain the Custom Position Of the Label While Resizing the Node

We can retain the label's offset value using the **SizeChanged** event. While resizing a node, the **SizeEvent** gets fired. Using this event we can retain the label's position.

```
[C#]

// Adding Event Handler
((DocumentEventSink)diagram1.Model.EventSink).SizeChanged += new
SizeChangedEventHandler(Form1_SizeChanged);
outerRect.Labels.Add(new Syncfusion.Windows.Forms.Diagram.Label());
outerRect.Labels[0].Text = "Rectangle";
outerRect.Labels[0].Position = Position.Custom;
outerRect.Labels[0].OffsetX = outerRect.Size.Width / 2;
outerRect.Labels[0].OffsetY = outerRect.Size.Height;

// Resizing
void Form1_SizeChanged(SizeChangedEventArgs evtArgs)
{
   outerRect.Labels[0].OffsetX = outerRect.Size.Width / 2;
```

```
   outerRect.Labels[0].OffsetY = outerRect.Size.Height;
}
```

```
[VB.NET]

' Adding Event Handler
AddHandler (CType(diagram1.Model.EventSink,
DocumentEventSink)).SizeChanged, AddressOf Form1_SizeChanged
outerRect.Labels.Add(New Syncfusion.Windows.Forms.Diagram.Label())
outerRect.Labels(0).Text = "Rectangle"
outerRect.Labels(0).Position = Position.Custom
outerRect.Labels(0).OffsetX = outerRect.Size.Width / 2
outerRect.Labels(0).OffsetY = outerRect.Size.Height

' Resizing
Private Sub Form1_SizeChanged(ByVal evtArgs As
Syncfusion.Windows.Forms.Diagram.SizeChangedEventArgs)
   outerRect.Labels(0).OffsetX = outerRect.Size.Width / 2
   outerRect.Labels(0).OffsetY = outerRect.Size.Height
End Sub
```

5.28 How To Retrieve the Port Information Of a Particular Symbol

You can retrieve port information of a particular symbol using the **HandlesHitTesting.GetConnectionPointAtPoint(Node, Point)** method.

This method has two parameters: *Node* and *Port*.

- Node specifies the symbol in which the port resides
- Point specifies the Point object that holds the location of the port.

```
[C#]

ConnectionPoint port =
HandlesHitTesting.GetConnectionPointAtPoint(circle, new Point(120,
120));
```

```
[VB.NET]
```



```
Dim port As ConnectionPoint =
HandlesHitTesting.GetConnectionPointAtPoint(circle, New Point(120,
120))
```

5.29 How To Serialize the Custom Property Of a Node

Essential Diagram supports custom serialization. To serialize the custom property, you should derive the **Group** class and create a custom node. After creating a custom node, you should override the **GetObjectData()** method and add the custom property in the SerializationInfo. This is illustrated in the below code snippet.

[C#]

```
[Serializable()]
public class CustomNode : Group
{
   protected CustomNode(SerializationInfo info, StreamingContext
   context) : base(info, context)
   {
      this.m_nodeInformation = info.GetString("strDescription");
   }
   protected override void GetObjectData(SerializationInfo info,
   StreamingContext context)
   {
      base.GetObjectData(info, context);

      // Additional member variables are serialized here.
      info.AddValue("strDescription", this.NodeInformation);
   }
}
```

[VB.NET]

```
<Serializable()>
Public Class CustomNode
Inherits Group
   Public Sub New()
   End Sub
   Protected Sub New(ByVal info As SerializationInfo, ByVal context As
   StreamingContext)
      MyBase.New(info, context)
      Me.m_nodeInformation = info.GetString("strDescription")
```

```
    End Sub
    Protected Overrides Sub GetObjectData(ByVal info As
    SerializationInfo, ByVal context As StreamingContext)
        MyBase.GetObjectData(info, context)

        ' Additional member variables are serialized here.
        info.AddValue("strDescription", Me.NodeInformation)
    End Sub
End Class
```

5.30 How To Set the Custom Position For a Label

We can adjust the label position by setting the **Position** property as 'Custom'. Then, we have to set the Offset values for the X and Y coordinates to specify the label position.

[C#]

```
// Setting custom position for a label
outerRect.Labels.Add(new Syncfusion.Windows.Forms.Diagram.Label());
outerRect.Labels[0].Text = "Rectangle";
outerRect.Labels[0].Position = Position.Custom;
outerRect.Labels[0].OffsetX = 50;
outerRect.Labels[0].OffsetY= 65;
```

[VB.NET]

```
' Setting custom position for a label
outerRect.Labels.Add(New Syncfusion.Windows.Forms.Diagram.Label())
outerRect.Labels(0).Text = "Rectangle"
outerRect.Labels(0).Position = Position.Custom
outerRect.Labels(0).OffsetX = 50
outerRect.Labels(0).OffsetY= 65
```

5.31 How To Protect Links From User Selection / Manipulation

You can protect links from user selection / manipulation by making use of the **EditStyle** class. By setting the **Enabled** property of the EditStyle class to *False*, you can disable the selection of a node link.

```
[C#]

//Creating Line connector.
LineConnector conn = new LineConnector(new PointF(1, 1), new PointF(2, 2));

//Disabling selection of the line connector.
conn.EditStyle.Enabled = false;
```

```
[VB.NET]

'Creating Line connector.
Dim conn As LineConnector = New LineConnector(New PointF(1, 1), New PointF(2, 2))

'Disabling selection of the line connector.
conn.EditStyle.Enabled = False
```

5.32 How To Smooth-out the Edges Of the Shapes In a Diagram

You can use the **Diagram.Model.RenderingStyle.SmoothingMode** property to smooth-out the edges, lines and curves of the shapes in a diagram.

```
[C#]

this.diagram1.Model.RenderingStyle.SmoothingMode = System.Drawing.Drawing2D.SmoothingMode.HighQuality;
```

```
[VB.NET]

Me.diagram1.Model.RenderingStyle.SmoothingMode = System.Drawing.Drawing2D.SmoothingMode.HighQuality
```

5.33 How to Move Nodes on a Diagram Programmatically?

You can move the desired collection of nodes to a diagam using the **MoveNodes** method. The following code example illustrates this.

[C#]

```
//Move the selected nodes by 20 pixels in both horizontally and vertically.
diagram1.MoveNodes(diagram1.Controller.SelectionList, 20, 20, MeasureUnits.Pixel);
```

[VB.NET]

```
'Move the selected nodes by 20 pixels in both horizontally and vertically.
diagram1.MoveNodes(diagram1.Controller.SelectionList, 20, 20, MeasureUnits.Pixel)
```

5.34 How to Remove the Gray Area around a Diagram?

You can remove the unnecessary gray area added around a diagram by setting the **ScrollVirtualBounds** property as **Empty**. The following code example illustrates this.

[C#]

```
//Remove the unwanted gray area around a diagram.
diagram1.View.ScrollVirtualBounds = RectangleF.Empty;
```

[VB.NET]

```
'Remove the unwanted gray area around a diagram.
diagram1.View.ScrollVirtualBounds = RectangleF.Empty;
```

5.35 How to Detect whether a Property Value is Changed in the Property Editor?

You can use the **PropertyValueChanged** event to detect whether a property value is changed in the property editor. The following code example illustrates this.

[C#]

```
//PropertyValueChanged event.
propertyEditor1.PropertyGrid.PropertyValueChanged += new
PropertyValueChangedEventHandler(PropertyGrid_PropertyValueChanged);


private void PropertyGrid_PropertyValueChanged(object s,
PropertyValueChangedEventArgs e)
{
     //Here, capture the changed property (e.ChangedItem) and process
the same.
}
```

[VB.NET]

```
'PropertyValueChanged event.
AddHandler propertyEditor1.PropertyGrid.PropertyValueChanged, AddressOf
PropertyGrid_PropertyValueChanged

Private Sub PropertyGrid_PropertyValueChanged(ByVal s As Object, ByVal
e As PropertyValueChangedEventArgs)
       'Here, capture the changed property (e.ChangedItem) and process
the same.
End Sub
```

5.36 How to Add Dynamically Created Symbol Palette to PaletteGroupBar

The following code sample demonstrates how to add a dynamically created symbol palette to the PaletteGroupBar control.

[C#]

```
            //Create an instance of SymbolPalette.
```

```
            SymbolPalette palette = new SymbolPalette("Palette");
            //Add some nodes to the palette.
            palette.AppendChild(new Ellipse(0, 0, 100, 100));
            palette.AppendChild(new RoundRect(new RectangleF(0, 0, 100,
50),
            MeasureUnits.Pixel));
            palette.AppendChild(new LineConnector(PointF.Empty, new
PointF(0,
            100)));
            //Add the palette to PaletteGroupBar.
            paletteGroupBar1.AddPalette(palette);
```

```
[VB.NET]


            'Create an instance of SymbolPalette.
            Dim palette As SymbolPalette = New SymbolPalette("Palette")
            'Add some nodes to the palette.
            palette.AppendChild(New Ellipse(0, 0, 100, 100))
            palette.AppendChild(New RoundRect(New RectangleF(0, 0, 100,
50),
            MeasureUnits.Pixel))
            palette.AppendChild(New LineConnector(PointF.Empty, New
PointF(0,
            100)))
            'Add the palette to PaletteGroupBar.
            paletteGroupBar1.AddPalette(palette)
```

5.37 How to Get the Undo/Redo Description?

The **HistoryManager** class has an option to get undo or redo description. You can use the **GetUndoDescriptions** or **RedoDescriptions** method to get the desired depth of descriptions available in the history of undo or redo list, respectively. The following code example illustrates this.

```
[C#]


string[] strDescriptions;
int nDescWanted = 1;
//Gets an undo description from the undo list.
int nDescReturned =
diagram1.Model.HistoryManager.GetUndoDescriptions(nDescWanted, out
strDescriptions);
```

```
//Gets a redo description from the redo list.
nDescReturned =
diagram1.Model.HistoryManager.GetRedoDescriptions(nDescWanted, out
strDescriptions);
```

```
[VB.NET]


Dim strDescriptions() As String
Dim nDescWanted As Integer = 1
'Gets an undo description from the undo list.
Dim nDescReturned As Integer =
diagram1.Model.HistoryManager.GetUndoDescriptions(nDescWanted,
strDescriptions)
'Gets a redo description from the redo list.
nDescReturned =
diagram1.Model.HistoryManager.GetRedoDescriptions(nDescWanted,
strDescriptions)
```

5.38 How to Import Visio Stencil?

You can import Visio stencils (.vss and .vsx) into symbol palettes. Essential Diagram uses the Visio stencil converter to convert the stencils as the symbol palette. You have to add **Syncfusion.Diagram.Utility.Windows.dll** as a reference in your application to use this converter.

The following code example illustrates how to convert Visio stencils into symbol palettes.

```
[C#]
SymbolPalette paletteToReturn = null;
string strFileName = "Basic Flowchart Shapes.vss";
//Create an instance of VisioStencilConverter
VisioStencilConverter converter = new
VisioStencilConverter(strFileName);
converter.ShowProgressDialog = true;
//Convert the stencil as SymbolPalette
paletteToReturn = converter.Convert();
paletteGroupBar1.AddPalette(paletteToReturn);
```

```
[VB.NET]
```



```
Dim paletteToReturn As SymbolPalette = Nothing
Dim strFileName As String = "Basic Flowchart Shapes.vss"
'Create an instance of VisioStencilConverter
Dim converter As VisioStencilConverter = New
VisioStencilConverter(strFileName)
converter.ShowProgressDialog = True
'Convert the stencil as SymbolPalette
paletteToReturn = converter.Convert()
paletteGroupBar1.AddPalette(paletteToReturn)
```

 ***Note: You must have Visio installed in your machine to import the stencils.***

5.39 How to Get a Node at a Point or under a Mouse Location?

GetNodeAtPoint

You can use the **GetNodeAtPoint** method of **Controller** to get a node at a point. You have to convert the point to a model coordinate before using this method in order to get the exact result when a diagram document is panned and zoomed.

[C#]

```
//Client location.
Point pt = new Point(200,150);
//Covert client location to model location.
pt = diagram1.Controller.ConvertToModelCoordinates(pt);
//Get the node in model coordinates.
Node node = diagram1.Controller.GetNodeAtPoint(pt);
```

[VB.NET]

```
'Client location.
Dim pt As Point = New Point(200,150)
'Covert client location to model location.
pt = diagram1.Controller.ConvertToModelCoordinates(pt)
'Get the node in model coordinates.
Dim node As Node = diagram1.Controller.GetNodeAtPoint(pt)
```

GetNodeUnderMouse

You can use the **GetNodeUnderMouse** method of **Controller** to get a node at a current mouse point. You don't have to convert the mouse point to a model coordinate while using this method. You just need to pass the current mouse point to the model coordinate.

[C#]

```
private void diagram1_MouseMove(object sender, MouseEventArgs e)
{
     //Gets the top node under the current mouse location.
     INode node = diagram1.Controller.GetNodeUnderMouse(e.Location);
}
```

[VB.NET]

```
Private Sub diagram1_MouseMove(ByVal sender As Object, ByVal e As
MouseEventArgs)
       'Gets the top node under the current mouse location.
       Dim node As INode =
diagram1.Controller.GetNodeUnderMouse(e.Location)
End Sub
```


Index

