Syncfusion Inc.

Corporate headquarters

9001, Aerial Center Parkway
Suite 110
Morrisville, NC 27560

Phone: *(919) 481 1974*
Fax: *(801) 640 9181*

Email: *info@syncfusion.com*

Technical Support

Direct-Trac Support System: *http://www.syncfusion.com/support*
Forums: *http://www.syncfusion.com/forums*
Email: *support@syncfusion.com*

Sales

Phone: *(919) 481 1974*
Toll Free: *(888) 9DOTNET*

Fax: *(801) 640 9181 (US & rest of the world except Europe)*
Fax: *+49(69)791249766 (Europe)*

Email: *sales@syncfusion.com*

Essential Diagram users guide 5
1 Introduction 5
1.1 Features 5
1.2 Software Requirements 7
1.3 Resources 7
1.3.1 User´s Guide 7
1.3.2 Class Reference 7
1.3.3 Samples 7
1.3.4 Knowledgebase 7
1.3.5 Technical Support 8
1.3.6 Forums 8
1.4 Licensing Information 8
1.5 Conventions Used In This Manual 8
2 Getting Started 8
2.1 Adding Controls and Components to the Toolbox 9
2.2 Debugging Support via the Assembly Manager 10
2.3 Distributing Essential Diagram 10
3 Design Overview 10
4 Conceptual Overview 10
4.1 Nodes and Hierarchies 10
4.2 Coordinate Systems 11
4.2.1 World Coordinates 12
4.2.2 View Coordinates 12
4.2.3 Device Coordinates 12
4.3 Models 13
4.4 Views 13
4.5 Controllers 14
4.6 Shapes 14
4.7 Symbols 14
4.7.1 Ports 14
4.7.2 Connections 15
4.7.3 Links 16
4.7.4 Labels 17
4.7.5 Symbol Palettes and Symbol Models 17
4.8 Properties and Styles 18
4.8.1 Property Containers 18
4.8.2 Late-Binding to Properties 18
4.8.3 Property Inheritance 21
4.8.4 Styles 22
4.9 Commands and Undo/Redo 22
4.10 Layers 22
4.11 Diagrams 23
5 Quick Start 26
5.1 Add a Diagram Control to a Form 26
5.2 Add a Symbol Palette Control 27
5.3 Add Controls to Interact with the Diagram 28
5.4 Activate User-Interface Tools 30
5.5 Add a PropertyEditor Control 31
5.6 Saving and Loading the Diagram 31
6 Using the Framework 32
6.1 Event Model 32
6.1.1 Event Routing 32

6.1.2 Example Event Handler 33
6.2 Diagram Serialization 36
6.3 Symbol Palette Serialization 37
6.4 Graph Navigation 39
6.5 Subclassing Symbols 39
6.6 Sub-classing the Diagram Control 41
6.7 Logical Units 42
6.8 Printing 43
7 Using the Symbol Designer 44
7.1 Where to find the Symbol Designer 44
7.2 User Interface Overview 44
7.3 Creating a new Symbol Palette 47
7.4 Adding Symbols to a Palette 48
7.5 Symbols versus Symbol Models 49
7.6 Editing Symbol Models with the Symbol Designer 49
7.6.1 Drawing Tools 49
7.6.2 Rotate Tools 50
7.6.3 Node Tools 50
7.6.4 Align Tools 51
7.6.5 Layout Tools 52
7.6.6 Nudge Tools 52
7.6.7 View Tools 53
7.7 Using the Property Editor 53
7.8 Saving and Loading Symbol Palettes 53
7.9 Symbol Size 54
7.10 Adding Labels 55
7.11 Adding Ports 56
7.12 Binding to a Sub-classed Symbol 56

Essential Diagram users guide

1 Introduction

Thank you for choosing Essential Diagram. Before getting started with the library,please spend some time reading through this manual. It provides an overview of the design and structure of the Essential Diagram framework and has information that you´ll need to get started using the product. Familiarity with this manual will help you get the most out of the product.

Essential Diagram is a native .NET UI library for creating interactive diagramming applications. Essential Diagram can be used with any .NET language including C#, VB.NET, and managed C++. Essential Diagram is designed for ease of use, flexibility, and high performance. It can be used to create a wide range of applications.Listed below are examples of applications that can be built using Essential Diagram.

- Flowcharting
- Workflow modeling
- Telecommunications network visualization
- Software engineering
- Architectural, engineering, and construction
- Commercial interior design
- Data visualization
- Simulation
- Electrical circuit and computer chip design

Please note that detailed documentation of the classes in the framework can be found in the *Essential Diagram Class Reference*

1.1 Features

The list of features in this section provides a quick overview of what Essential Diagram is and its capabilities.

- Retained-mode interface for 2D interactive graphics
- Hierarchical node structure supporting nesting
- Model-view-controller architecture
    - Clearly separates data, presentation, and user interaction into separate model, view, and controller components
    - Supports extending model, view, and controller components
    - Mix and match model, view, and controller components for custom functionality
- Matrix transformations
    - Translate
    - Rotate
    - Scale
- Shape nodes

    - Rectangle
    - Rounded Rectangle
    - Ellipse
    - Line
    - PolyLine
    - Curve
    - ClosedCurve
    - RoundRectangle
- Images
    - Bitmaps with support for multiple formats
    - Enhanced metafiles
- Text
    - Font properties
    - Text editing
    - Text rotation
- Symbol nodes
    - Container for child nodes
    - Text labels
    - Ports and Connections
- Link nodes
    - Specialized symbols
    - Contain a line shape and ports on either end
- Command architecture
    - Undo/redo support
    - Macro commands
- Interactive shape drawing tools
- Vertex editing
- Group and ungroup nodes
- Hit testing
- Coordinate conversion
- Zooming
- Grid with snap to grid feature
- Real-world logical units (e.g. Metric units, English units)
- Plug-in architecture for interactive tools
- Print and print preview
- Diagram control
    - WinForm control that can be added to the Visual Studio .NET toolbox
    - Container for model, view, and controller objects
- PaletteGroupBar control
    - WinForm control that can be added to the Visual Studio .NET toolbox
    - Displays list symbols in a symbol palette as icons
    - Allows user to drag and drop symbols onto diagrams
    - Supports multiple symbol palettes at a time
    - User interface similar to Microsoft Outlook bar
    - Based on Syncfusion GroupBar and GroupView controls
- PropertyEditor control
    - WinForm control that can be added to the Visual Studio .NET toolbox
    - Allows user to view and edit the properties of one or more nodes in a diagram
    - Based on Microsoft .NET PropertyGrid control
- Symbol Designer utility for creating palettes of symbols
- Samples
    - Generic diagramming tool

- Circuit designer
- Workflow editor
- Flowcharting tool

1.2 Software Requirements

Essential Diagram is compatible with Microsoft Visual Studio .NET. Please note that it is not compatible with the beta and release candidate version of Microsoft Visual Studio .NET. Essential Diagram controls and components can be used in any .NET environment, including C#, VB.NET, and managed C++. Essential Diagram is supported on all Windows Platforms supported by the Microsoft .NET platform.

1.3 Resources

This section provides information about various resources available to Essential Diagram customers. It is our foremost goal to ensure that Essential Diagram can be used easily and effectively.

1.3.1 User´s Guide

This document explains the features and design of the product. It shows how the various features can be used through the designer and through code. Code snippets are provided in C# along with detailed instructions on usage.

1.3.2 Class Reference

The Essential Diagram Class Reference provides detailed documentation of the classes in Essential Diagram. The Class Reference is integrated into the Visual Studio help system to facilitate easy access and use. If you have questions about the usage of a particular control, class, or method, the class reference is the place to look.

1.3.3 Samples

Essential Diagram ships with samples that demonstrate its use in both Visual Basic .NET and C#. The samples can be found under the [InstallDir]/Essential Suite/Diagram/Samples directory.

1.3.4 Knowledgebase

The Essential Diagram knowledge base is available on-line at:

http://www.syncfusion.com/kb/Diagram/

The knowledgebase is a dynamic document that addresses frequently asked questions about Essential Diagram. This would be the first place to look for solutions to problems and answers to questions that you may have about the product.

1.3.5 Technical Support

At Syncfusion, we pride ourselves on our support infrastructure. We believe that our support infrastructure allows us to provide a level of service that is unmatched in the software tools industry.

Technical support can be obtained at any time using the Syncfusion Direct-Trac Developer Support System available at:

http://www.syncfusion.com/support

Support incidents can be created and tracked to completion 24 hours a day, 7 days a week. We guarantee same business day turn around for incidents that are entered into the Direct-Trac support system before 2:00 PM on any business day. For incidents created or updated after 2 PM, we guarantee next business day turn around.

1.3.6 Forums

1.4 Licensing Information

Syncfusion has a simple, royalty-free licensing model. Components are licensed to a single user. We recognize that you often work at home or on your laptop in addition to your work machine. Therefore, our license permits our products to be installed on a laptop and a home machine in addition to a work machine for your use.

1.5 Conventions Used In This Manual

The following typographical conventions are used to help identify important items in the text.

Example	**Description**
Menu|Menu Item	Bold font and pipe symbol used for menu items.
Expression	Words in italics indicate placeholders for information you must supply, such as a variable or parameters for methods
for (i = 0; i < n; i++)	The font Courier New is used for source code
Shape	Class names are bold
INode	Interface names are bold, italic

2 Getting Started

This section contains information on getting started with Essential Diagram. It explains how to add the Essential Diagram components to the Visual Studio .NET

toolbox, how to switch between "Debug" and "Release" version of the library, and how to distribute Essential Diagram with your applications.

2.1 Adding Controls and Components to the Toolbox

Open the Visual Studio .NET Toolbox window and select "Add Tab" using the context menu. Set the name of the new tab to "Syncfusion". With this tab active, select "Customize Toolboxâ€¦" from the context menu. This will open the "Customize Toolbox" folder. Select the ".NET Framework Components" tab. All installed Syncfusion design-time controls should appear in this list. Make sure that the check box next to the entries that you want to use is checked. This will cause them to be included in the toolbox.

The table below lists the Essential Diagram components.

Control	**Namespace**
Diagram	Syncfusion.Windows.Forms.Diagram.Controls
PaletteGroupBar	Syncfusion.Windows.Forms.Diagram.Controls
PaletteGroupView	Syncfusion.Windows.Forms.Diagram.Controls
PropertyEditor	Syncfusion.Windows.Forms.Diagram.Controls

It is recommended that you keep the Syncfusion components on a separate tab in the toolbox for easy access.

Note: If you do not see the Syncfusion controls listed in the ".NET Framework Components" tab, click on the "Browse" button and select the Synfusion.Diagram.Controls assembly. This will manually add the Essential Diagram design-time controls to the list. Then select the controls as described above. Please report the problem by submitting a support incident at *http://www.syncfusion.com/support,* and include system details to help us reproduce and fix the problem.

2.2 Debugging Support via the Assembly Manager

2.3 Distributing Essential Diagram

3 Design Overview

Essential Diagram is designed to be modular, easy to use, easily extended, and fast. Coupling between components is kept to minimum, which allows low-level components to be used independently of higher level components and controls. For example, you can bypass using the **Diagram** control and use lower level components such as the **Model, View** and **Controller** to create interactive 2D graphics applications. Another example of low coupling is the **Shape** class, which can render itself onto any **System.Windows.Forms.Graphics** object independently of the other Essential Diagram components.

The library consists of two assemblies: Essential.Diagram.dll and Essential.Diagram.Controls.dll. The Essential.Diagram.dll assembly contains the core classes and components. The Essential.Diagram.Controls.dll assembly contains a **Diagram** control that can be installed in the Microsoft Visual Studio .NET toolbox, as well other WinForm user interface controls such the **SymbolPalette** control.

One of the central design concepts of Essential Diagram is the model-view-controller (MVC) design pattern. This design pattern provides a clear separation between the data, the view of the data, and the user interface. A model contains the nodes that are rendered onto a view. The model contains the data that describes a diagram. The view represents a rectangular area of the screen in which a model is rendered. The controller receives input from the user and translates it into actions and commands that are applied to the model and view.

The cool thing about using MVC in Essential Diagram is that models, views, and controllers can be mixed and matched. This is especially useful when customizing the user interface. Developers can write their own custom controllers and use them instead of the stock controllers provided with the library.

4 Conceptual Overview

This section explains the key concepts of the Essential Diagram framework. After reading this section you should have a good understanding of the terminology and structure of the framework.

4.1 Nodes and Hierarchies

An important concept in Essential is that of a node. A node is a named object in a hierarchy of objects. A node has a parent and a name that is unique within the scope of its parent. The fully qualified name of a node includes the names of the node´s ancestors separated by the ´.´ delimiter. In other words, it´s a named object hierarchy. A simple or leaf node has parent, but it has no children. Composite nodes may have zero or more children.

All node objects implement the **INode** interface. Nodes that contain children implement the **ICompositeNode** interface, which derives from **INode**. Shapes, text, images, and ports are examples of simple, leaf nodes that implement the INode interface. Models, symbols, and groups are examples of composite nodes that implement the **ICompositeNode** interface. Figure 1 shows an example node hierarchy.

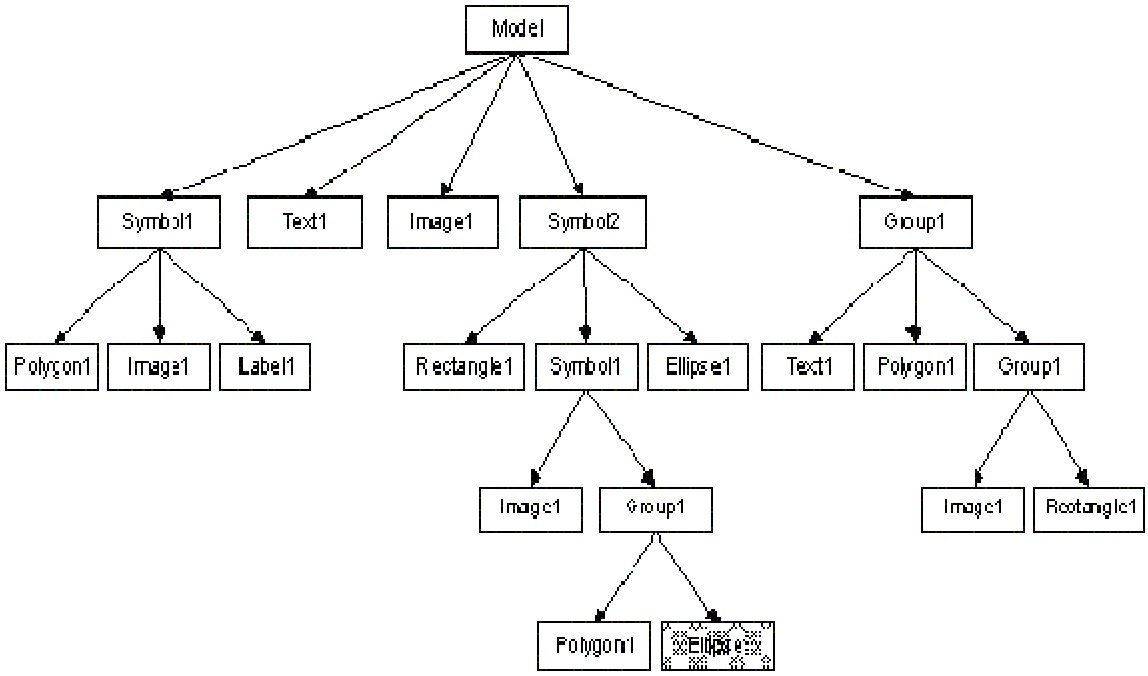

Figure 1. 1 Example node hierarchy.

Node names can be assigned by the user or programmatically by your application. The name of a node must be unique within the scope of its parent. If a node does not already have a unique name at the time it is inserted into the hierarchy, it is assigned a default name based on its type and an integer suffix. The fully qualified name of a node is always unique within the hierarchy. For example, the fully qualified name of the shaded node in Figure 1 is

Model.Symbol2.Symbol1.Group1.Ellipse1.

4.2 Coordinate Systems

A coordinate system provides a way to plot points in space relative to a fixed origin. In a two-dimensional coordinate space, the horizontal line that intersects the origin is referred to as the X-axis and the vertical line that intersects the origin is referred to

as the Y-axis. Each coordinate system defines its own unit of measure for plotting the distance between a given point and the origin. Any given point in the coordinate space is defined by two values: its distance from the origin along the X-axis and its distance from the origin along the Y-axis. A two-dimensional coordinate system is also referred to as a Cartesian coordinate system, which you are undoubtedly familiar with from middle school algebra. Essential Diagram uses three different coordinate systems: world coordinates, view coordinates, and device coordinates.

4.2.1 World Coordinates

The world coordinate system defines coordinates in terms that are useful to the application and meaningful to the end user. The unit of measure for world coordinates can be pixels, inches, millimeters, or points and depends on the application. In Essential Diagram, world coordinates are stored as floating point values and the unit of measure is one of the values from the **System.Drawing.GraphicsUnit** enumeration.

4.2.2 View Coordinates

The view coordinate system maps world coordinates onto a viewing area. Think of it as a logical window that sits on top of the world coordinate system and can be moved around to view different parts of the world coordinate system. The unit of measure for view coordinates is the same as the world coordinate space that is mapped onto the view coordinate space.

The origin in the view coordinate system is a point somewhere in the world coordinate space. As the origin of the view coordinate system changes, the mapping between world and view coordinates changes. For example, if the origin of the view is set to the world coordinates (5,10), then the mapping the world point (5,10) to the view coordinate system results in (0,0). Mapping the world point (10,5) to the view coordinate system results in (5,-5).

The view coordinate system can also scale world coordinates using a magnification value. This allows the viewing area to zoom in and out on the world coordinate system.

4.2.3 Device Coordinates

The screen of a computer or a window on the screen defines a two-dimensional coordinate space in which points are plotted in units of pixels relative to an origin that is defined to be the upper-left hand corner of the screen or window. The coordinate space used to render points to a printer also uses pixels as the unit of measure with the origin at the upper-left hand corner of the page. The important difference between a computer screen and a printer is the device resolution or DPI (dots per inch). The resolution of a computer screen is typically around 96 dpi and a printer is typically 600dpi or greater. This means that drawing a line 96 pixels in length on a screen that is 96 dpi will result in a line that measures approximately 1 inch. Drawing the same 96 pixel line on a 600 dpi printer will result in a line that is only a fraction of an inch - 96 pixels / 600 pixels per inch = 0.16 inches to be exact. The important thing to keep in mind is that pixels are very device specific units of measure that have no constant conversion to any real world measurement.

The device coordinate space defines coordinates in terms of output devices like computer screens and printers. The unit of measure for coordinates is pixels and the origin is defined to be the upper-left hand corner of the drawing surface. Device coordinates are mapped to and from view coordinates. The mapping of view coordinates to device coordinates depends on two things: unit of measure in the view coordinate space and the resolution of the device.

4.3 Models

A model is a collection of nodes that are rendered onto a view and manipulated by a controller. It represents the data or document portion of a diagram. The **Model** class provides the base implementation for all models.

Nodes in the model are accessed, added, and removed through the **ICompositeNode** interface, which the **Model** class implements. The **Model** also exposes a Nodes property which can be used be for the same purpose. Another way to add nodes a model is to execute an **InsertNodesCmd** through the controller. The code shown below creates a new rectangle and adds it to a model.

C# code

```
 public void AddRect(Model mdl)
{
        Syncfusion.Windows.Forms.Diagram.Rectangle rect;
        rect = new Rectangle(20,10,80,80);
        rect.FillStyle.Color = Color.Green;
        mdl.AppendChild(rect);
}
```

Another interesting and occasionally useful aspect of models is that they can be nested. Since models implement the **INode** interface, they can be added to other composite nodes.

The model is responsible for firing certain events that occur within the object hierarchy, such as a node being added or moved.

4.4 Views

A view is an object that draws a model onto a rectangular area of the screen. The **Syncfusion.Windows.Forms.Diagram.View** class provides the base class implementation for all views. You can use this class directly or you can derive new classes from it. A view is hosted inside of a **System.Windows.Forms.Control** object and renders itself onto the control.

Views can also render additional visual information that does not exist inside the model such as bounding boxes, coordinate axes, and grids. These additional view specific objects are referred to as decorators because they provide additional visual aids and window dressing to the view, but are not actually part of the model.

In addition to rendering the model onto the screen, the view is responsible for conversions between world, view, and device coordinates. The view maps world coordinates onto view coordinates by applying a matrix transformation that is loaded with the view origin and the magnification (zooming) factor. The view origin is the offset (i.e. translate) value in the transformation matrix. The magnification factor is the scaling value in the transformation matrix. The result is a transformation matrix that scrolls and zooms world coordinates to the correct place in the view.

Views also provide methods for performing hit testing. When a mouse event occurs, the point at which the event occurred is always given in device coordinates. Since the view knows how to map device coordinates to view and world coordinates, it is the logical entry point for hit testing routines.

4.5 Controllers

A controller is an object that receives user input and translates it into actions and commands on the model and view. A controller contains one or more tools, which are reusable objects that implement a slice of functionality for the controller. Tools are reusable because the same tool can be used in many different controllers.

Controllers can receive input in several ways. They can receive window events from a window, input from DirectInput devices, or animation events from an animation loop. Depending on the controller and the behaviors it exposes, a controller might receive input from one or more these sources.

4.6 Shapes

A shape is a type of node that contains a set of points and that renders those points. Shapes use the **System.Drawing.GraphicsPath** class for rendering and hit testing. A **GraphicsPath** is an object that contains a collection of points and one or more drawing instructions. There is a base **Shape** class from which specific types of shapes are derived such as **Rectangle, Ellipse, Curve,** and **Polygon.**

4.7 Symbols

A symbol is a composite node that supports connections to other symbols. The visual appearance of a symbol is determined by the child nodes it contains. Symbols may contain one or more ports, which are points on the symbol at which connections to other symbols can be established. Symbols may also contain zero or more labels, which are text objects that are positioned relative to a control point on the symbol.

4.7.1 Ports

A port is an object anchored to a symbol that provides a location at which connections to other symbols can be made. There are several types of ports, each of which differ 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 crosshairs. Another example is a **CenterPort**, which always positions itself in the center of the symbol and has no visual representation.

4.7.2 Connections

A connection is an object that binds together two ports. A connection consists of two ports: a source port and a target port. The distinction between the source port and target is not really significant, because a connection doesn´t have a direction. The names source and target could just as easily be Port1 and Port2.

Connections are used to navigate from a symbol to the symbols that it is connected to. They are also used to notify connected symbols of certain events, such as the symbol moving. Symbols that are notified of a connected symbol moving can take action, such as repositioning itself or severing the connection.

The figure below demonstrates how symbols, ports, and connections are related. In the example, *Symbol 1* contains two ports and is connected to *Link 1* and *Symbol 3*. *Link 1* has two ports: a head port and a tail port. *Symbol 2* contains only one port, which is connected to *Link 1. Symbol 3* has two ports, one of which is connected to *Symbol 1*.

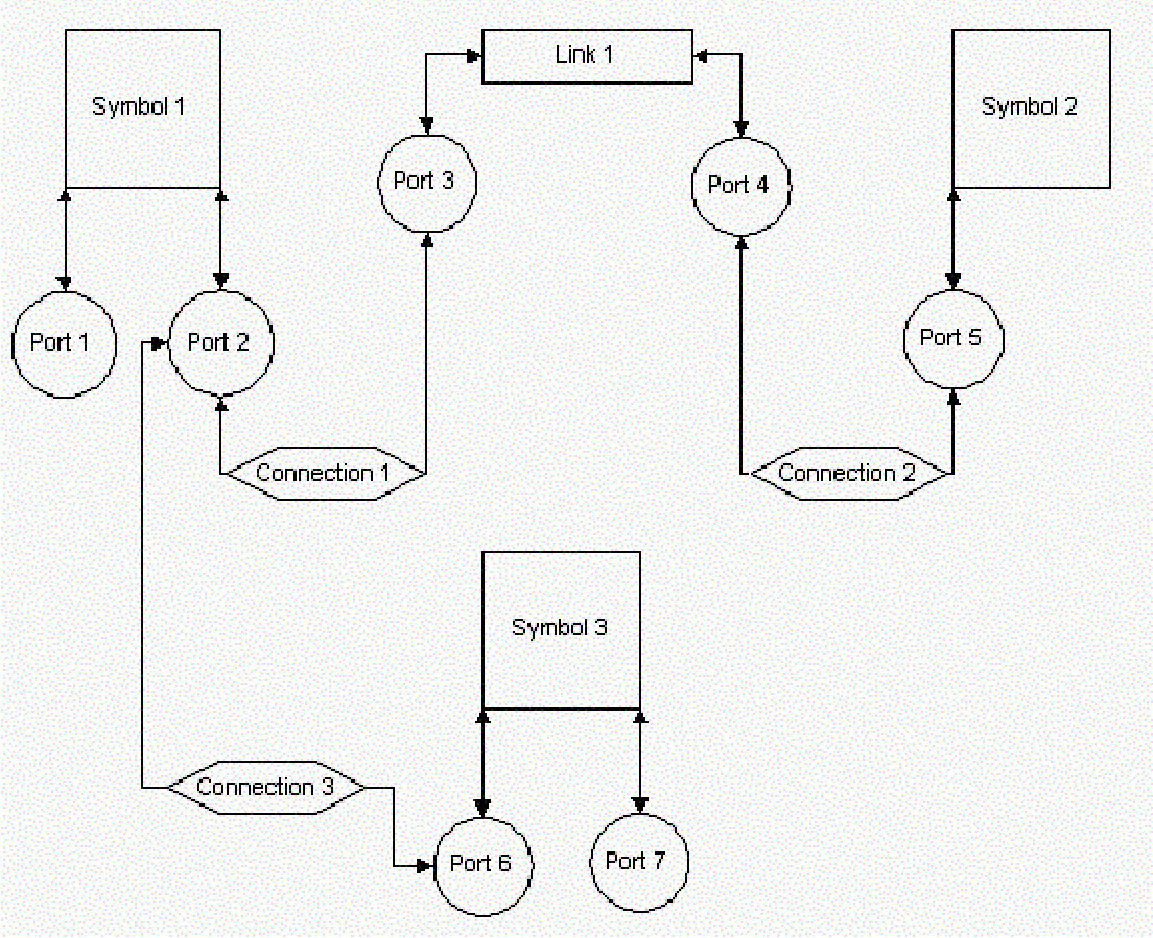

Figure 2. Connections.

4.7.3 Links

A link is a special type of symbol that has two ports: a head port and a tail port. Links contain a single child node referred to as the link shape. The tail port is attached to the first vertex in the link shape. The head port is attached to the last port in the link shape. The link shape can be any type of node that implements the IPoints interface and contains 2 or more points. The head and tail ports define a direction for a link. When the node hierarchy is viewed as a graph, links are treated as edges in the graph.

The figure below shows a diagram containing three symbols and two links.

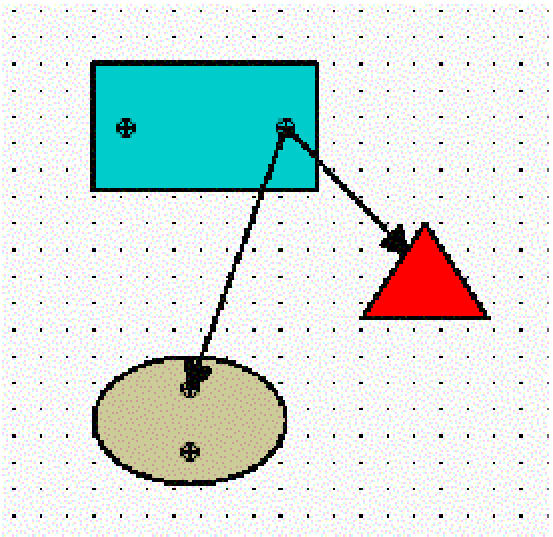

Figure 3. Links.

Notice that both links have their tail ports connected to the same port on the rectangular symbol. The link between the rectangle symbol and the triangle symbol has its head port connected to the implicit center port of the triangle. The reason that it stops at the boundary of the triangle is that the triangle´s center port has its AttachAtPerimeter property set to true. If it wasn´t, then the link would terminate at the center of the triangle symbol.

4.7.4 Labels

A label is a text node that is anchored to a symbol or a link. A label is positioned relative to the container (i.e. symbol or link) that owns it and always moves when its container moves. Scaling and rotating a label´s container does not affect the label. The type of container that the label is anchored to determines how it is positioned. Labels can be anchored to a control point on the container´s bounding box, with the possible control points being top-left, top-center, top-right, middle-left, center, middle-right, bottom-left, bottom-center, and bottom-right. Labels anchored to links can also be positioned based on a percentage of the distance between the head and tail points on the link.

4.7.5 Symbol Palettes and Symbol Models

Symbols can be created and inserted into Essential Diagram applications from a symbol palette, which is a special type of model that contains one or more symbol models. A symbol model is a node that describes a type of symbol that can be created. A symbol model can create instances of symbols that look and behave the same way. When a symbol model creates a new instance of a symbol, it clones all of the nodes and properties it contains and loads them into the new symbol instance. It can be thought of as a persistent symbol factory. Symbol palettes provide a way to group a collection of symbol models together.

4.8 Properties and Styles

Essential Diagram implements a property model that supports late-binding and inheritance. The **IPropertyContainer** interface provides the mechanism to support these features. It defines methods for storing, retrieving, adding, removing, and enumerating the properties contained by an object.

4.8.1 Property Containers

Essential Diagram implements a property model that supports late-binding and inheritance. The **IPropertyContainer** interface provides the mechanism to support these features. It defines methods for storing, retrieving, adding, moving, and enumerating the properties contained by an object.

4.8.2 Late-Binding to Properties

The term late-binding is used here to describe the ability of a piece of code to store and retrieve property values of objects without any compile-time knowledge of the object type. This is accomplished by using the **IPropertyContainer** interface to access properties instead of calling compiled get/set accessor methods.

The following example demonstrates the usefulness of late-binding for properties. It defines two classes, **Foo** and **Bar**, each of which contains a property named Weight. Both classes implement the **IPropertyContainer** interface, but their implementations differ based on how the properties are stored. The third class named **UseFooBar** contains a Main method that demonstrates the different ways in which the properties can be accessed.

C# code

```
class Foo : IPropertyContainer
{
// The Foo class declares a field at compile-time in which
// to store the Weight property

        private int weight;

        public Foo()
        {
                this.weight = 5;
        }

        public int Weight
        {
                get
                {
                        return this.weight;
                }
                set
                {
```

```
                this.weight = value;
            }
        }

        // NOTE: This is only a partial implementation of the
        // IPropertyContainer interface.

        public object GetPropertyValue(string propertyName)
        {
            if (propertyName == "Weight")
            {
                return this.Weight;
            }
            return null;
        }

        public void SetPropertyValue(string propertyName, object
value)
        {
            if (propertyName == "Weight")
            {
                this.Weight = value;
            }
        }
}

class Bar : IPropertyContainer
{
// The Bar class uses a Hashtable to store properties.

        private Hashtable props = new Hashtable();

        public Bar()
        {
            this.SetDefaultPropertyValues();
        }

        public int Weight
        {
            get
            {
                return this.GetPropertyValue("Weight");
            }
            set
            {
                this.SetPropertyValue("Weight", value);
            }
        }

        // NOTE: This is only a partial implementation of the
        // IPropertyContainer interface.

        public object GetPropertyValue(string propertyName)
        {
            if (this.props.Contains(propertyName))
            {
                return this.props[propertyName];
```

```
                }
                return null;
        }

        public void SetPropertyValue(string propertyName, object
value)
        {
                if (this.props.ContainsKey(propertyName))
                {
                        this.props[propertyName] = value;
                }
                else
                {
                        this.props.Add(propertyName, value);
                }
        }

        public void SetDefaultPropertyValues()
        {
                this.props.Add("Weight", 20);
        }
}

class UseFooBar
{
        static void Main()
{
        Foo foo = new Foo();
        Bar bar = new Bar();
        int sum;

                // Access Weight properties using early-binding
        sum = SumEarlyBinding(foo, bar);
        Console.WriteLine(sum.ToString());

                // Access Weight properties using late-binding
        sum = SumLateBinding(foo, bar);
        Console.WriteLine(sum.ToString());

}

// This method can only be used with objects of type
// Foo and Bar.
static int SumEarlyBinding(Foo foo, Bar bar)
{
        int w1 = foo.Weight;
        int w2 = bar.Weight;
        return w1 + w2;
}

// This method can be used with any two objects that implement
// the IPropertyContainer interface
static int SumLateBinding(IPropertyContainer c1,
  IPropertyContainer c2)
{
        int w1 = (int) c1.GetPropertyValue("Weight");
        int w2 = (int) c2.GetPropertyValue("Weight");
```

```
        return w1 + w2;
}
}
```

4.8.3 Property Inheritance

Objects that are organized in a hierarchical structure can implement **IPropertyContainer** in such a way as to support run-time inheritance. This means that if the object doesn´t contain a value for a given property, it can retrieve the value of the property from its parent. This is a very powerful technique that allows multiple child objects to share the properties of a common parent. It is more efficient in terms of storage space, because the child objects do not have to store values for every property they support. They only store values for properties that are explicitly assigned them and inherit values for properties that have not been explicitly set.

The node hierarchy inside of an Essential Diagram model uses this technique. The following code shows how the IPropertyContainer.GetPropertyValue method can be implemented in order to support inheritance.

C# code

```
 public virtual object GetPropertyValue(string propertyName)
{
        // NOTE: propertyValues is declared as a Hashtable
        if (this.propertyValues.Contains(propertyName))
        {
                return this.propertyValues[propertyName];
        }

        // NOTE: parent is declared as an INode
        if (this.parent != null)
        {
                IPropertyContainer parentProps;
parentProps = this.parent.GetPropertyContainer(this);
                if (parentProps != null)
                {
                        return
parentProps.GetPropertyValue(propertyName);
                }
        }

        return null;
}
```

This implementation of IPropertyContainer.GetPropertyValue first looks in a Hashtable maintained by the object. If the property is not found in the Hashtable, then the request is forwarded to the parent node.

4.8.4 Styles

The **Bar** class in the previous example code exposes the *Weight* property as a compile-time property that is a wrapper around the **IPropertyContainer** interface. In other words, the Weight property can be accessed either as Bar.Weight or more generically as Bar.GetPropertyValue("Weight"). The Bar.Weight property provides a convenient way to access the Weight property for code that doesn´t mind being bound to the **Bar** class at compile-time.

Styles work on the same concept. A style is an object that encapsulates a group of related properties in a property container and provides compile-time wrappers for those properties. Three commonly used styles in Essential Diagram are the **FontStyle, FillStyle,** and **LineStyle** classes. These classes (and all styles) are derived from the base **Style** class, which defines a reference to the **IPropertyContainer** that the style accesses. A style must be attached to a property container in order to work.

4.9 Commands and Undo/Redo

A command is an object that encapsulates an action and the data required to perform the action. All commands implement the **ICommand** interface, which defines methods to execute and undo the command. The implementation of a command class typically contains properties and/or methods for assigning data to the command that will be used when it is executed. Most commands in Essential Diagram operate on one or more nodes and derive from either the **SingleNodeCommand** or **MultipleNodeCommand** class.

One of the advantages of encapsulating commands as objects is that they can be stored and undone at a later time. This is the basis of Essential Diagram´s undo/redo feature. Commands are executed by an object that implements the **ICommandDispatcher** interface (typically the Controller). The command dispatcher manages two stacks: an undo stack for commands that can be undone and a redo stack for commands that can be re-executed. The **ICommandDispatcher** interface defines an UndoCommand method which removes the command from the top of the undo stack and calls its Undo method. It also defines a RedoCommand method, which removes the command from the top of the redo stack and calls the Do method on it.

4.10 Layers

It is often necessary to control the Z-order of nodes and to group nodes together that share common default properties. For example, you might want all text nodes in a diagram to share the same font, always be rendered on top of all other nodes, and be able to toggle the visibility of text on and off. You might want to do a similar thing with links, where all links share the same line properties and can be shown or hidden by toggling one property. Layers make it possible to accomplish these sorts of things.

A layer is a collection of nodes that share a common set of default properties and the same Z-order relative to other layers. The term Z-order refers to the order in which nodes are rendered on the diagram. Nodes that are rendered first are on the bottom

of the Z-order and can be obscured by nodes that are higher in the Z-order and rendered subsequently. A layer contains zero or more nodes and is responsible for rendering those nodes. Since the nodes in a layer are rendered as a group, their Z-order is the same relative to other layers in the diagram. For example, if layer A has a higher Z-order than layer B, then all nodes in layer B will be rendered behind those in layer A. If the Visible flag on a layer is set to false, none of the nodes in the layer will be rendered.

The nodes in the layer can inherit properties from the layer. If a property is not explicitly set in a node, the node inherits the property from the layer .If the layer does not have the property set, then the layer chains up to the model to get the property. This allows all nodes in a layer to share the same default properties.

Getting back to the example described earlier, separating text and links from other nodes can be accomplished by creating three layers: one for text, one for links, and one for all other nodes. In order to toggle the visibility of text and links, you would simply set the Visible property on the appropriate layer. Setting the font properties on the text layer would immediately change the font for all text nodes that do not have explicit font properties set. The same would be true of line properties in the link layer.

Layers are managed by the model. The **Model** class contains a Layers property that contains all layers in the diagram. Layers cannot be nested and they are not nodes in the node hierarchy. The **Model** class has a DefaultLayer property that determines which layer new nodes will be added to. A model must contain at least one layer.

4.11 Diagrams

The **Diagram** object is the top-level object in an Essential Diagram application. It combines the model, view, and controller into a single object that can be placed onto a form. The **Diagram** control has Model, View, and Controller properties that can be used to access the individual components. It also provides some convenient wrapper functions for calling commonly used methods on the model, view, and controller. The **Diagram** control is a window that paints itself by rendering the view in its available client area. It also implements scrolling, enabling horizontal and vertical scrollbars when the size of the model exceeds the size of the window.

The following example code shows a form that contains a diagram. All of the code shown in this example is generated by Visual Studio when you drag and drop a **Diagram** control from the toolbox onto a form.

C# code

```
using System;
using System.Drawing;
using System.Collections;
using System.ComponentModel;
using System.Windows.Forms;
using System.Data;

using Syncfusion.Windows.Forms.Diagram;
```

```
using Syncfusion.Windows.Forms.Diagram.Controls;

namespace UsersGuide
{
public class Form1 : System.Windows.Forms.Form
{
        private Diagram diagram1;

        ///
        /// Required designer variable.
        ///
        private System.ComponentModel.Container components = null;

        public Form1()
        {
                //
                // Required for Windows Form Designer support
                //
                InitializeComponent();
        }

        #region Windows Form Designer generated code
        ///
        /// Required method for Designer support - do not modify
        /// the contents of this method with the code editor.
        ///
        private void InitializeComponent()
        {
                this.diagram1 = new Diagram();
                this.SuspendLayout();
                //
                // diagram1
                //
                this.diagram1.AllowDrop = true;
                //
                // diagram1.Controller
                //
                this.diagram1.Controller.MaxHistory = 256;
                this.diagram1.Controller.SelectHandleMode =
SelectHandleType.Resize;
                this.diagram1.Dock =
System.Windows.Forms.DockStyle.Fill;
                this.diagram1.HScroll = true;
                this.diagram1.Location = new System.Drawing.Point(0,
0);
                //
                // diagram1.Model
                //
                this.diagram1.Model.BoundaryConstraintsEnabled =
true;
                this.diagram1.Model.Height = 5000F;
                this.diagram1.Model.Name = "Model";
                this.diagram1.Model.PageScale = 1F;
                this.diagram1.Model.PageUnit =
System.Drawing.GraphicsUnit.Pixel;
                this.diagram1.Model.Width = 5000F;
                this.diagram1.Name = "diagram1";
```

```
                this.diagram1.NudgeIncrement = 1F;
                this.diagram1.PropertyEditor = null;
                this.diagram1.ScrollGranularity = 0.5F;
                this.diagram1.Size = new System.Drawing.Size(416,
358);
                this.diagram1.TabIndex = 0;
                this.diagram1.Text = "diagram1";
                //
                // diagram1.View
                //
                this.diagram1.View.BackgroundColor =
System.Drawing.Color.DarkGray;
                this.diagram1.View.Grid.HorizontalSpacing = 10F;
                this.diagram1.View.Grid.MinPixelSpacing = 4;
                this.diagram1.View.Grid.SnapToGrid = true;
                this.diagram1.View.Grid.VerticalSpacing = 10F;
                this.diagram1.View.Grid.Visible = false;
                this.diagram1.View.HandleAnchorColor =
System.Drawing.Color.LightGray;
                this.diagram1.View.HandleColor =
System.Drawing.Color.White;
                this.diagram1.View.HandleSize = 6;
                this.diagram1.View.ShowPageBounds = true;
                this.diagram1.VScroll = true;
                //
                // Form1
                //
                this.AutoScaleBaseSize = new System.Drawing.Size(5,
13);
                this.ClientSize = new System.Drawing.Size(416, 358);
                this.Controls.Add(this.diagram1);
                this.Name = "Form1";
                this.SizeGripStyle =
System.Windows.Forms.SizeGripStyle.Show;
                this.Text = "Form1";
                this.ResumeLayout(false);

        }
        #endregion

        ///
        /// The main entry point for the application.
        ///
        [STAThread]
        static void Main()
        {
                Application.Run(new Form1());
        }
}

}  // end namespace
```

5 Quick Start

This section describes how to get started using Essential Diagram and how use some of the basic features of the framework.

Please note that the sample in this section uses very basic user interface elements for the sake of simplicity and clarity. A simple menu is used to activate commands that would typically be available on toolbars, and the subject of docking and layout is ignored. Please refer to the Essential Diagram samples included with the product for examples of how to add more sophisticated user interface elements in your application.

5.1 Add a Diagram Control to a Form

The easiest way to add a diagram to a form is to select it in the Visual Studio toolbox and drop it onto the form. Refer to *Section 2.1* for instructions on adding the Essential Diagram controls to the Visual Studio toolbox.

After adding the **Diagram** to a form, the form will look something like this.

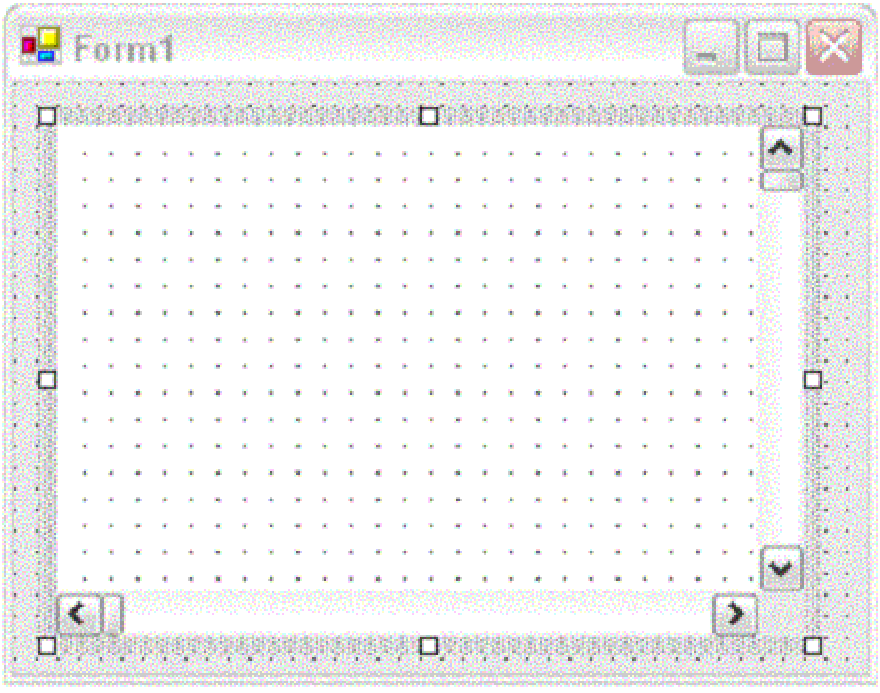

Figure 4. Diagram Control.

The Diagram control can be moved, resized or docked. Design-time properties of the diagram control can be edited in the property grid. The Model, View, and Controller

objects are available as design-time properties. The model, view, and controller objects can be expanded in the property grid to reveal more design-time properties.

5.2 Add a Symbol Palette Control

The **PaletteGroupBar** control provides a way for users to drag and drop symbols onto a diagram. It is based on the Syncfusion Tools **GroupBar** control. Each symbol palette loaded in the **PaletteGroupBar** occupies a panel that can be selected by a bar button that 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.

Follow the instructions below to add a PaletteGroupBar control to the form.

1. Select **PaletteGroupBar** control in the Visual Studio toolbox and drag it onto the form
2. Open *Properties* window and click on the GroupBarItems property. This will open the *GroupBarItem Collection Editor* dialog.
3. Click the *Add* button to add a new group bar item
4. Click *OK* to close the dialog
5. Click on the client area of the new group bar item to select the **PaletteGroupView** object
6. Go to the *Properties* window and click on the Palette property. This will open the *Open symbol palette* dialog
7. Browse to the [InstallDir]/Essential Suite/Diagram/Symbol Palettes directory and select the "Test Symbols.edp" palette file and then click OK.

With a little formatting, your form should now look something like this.

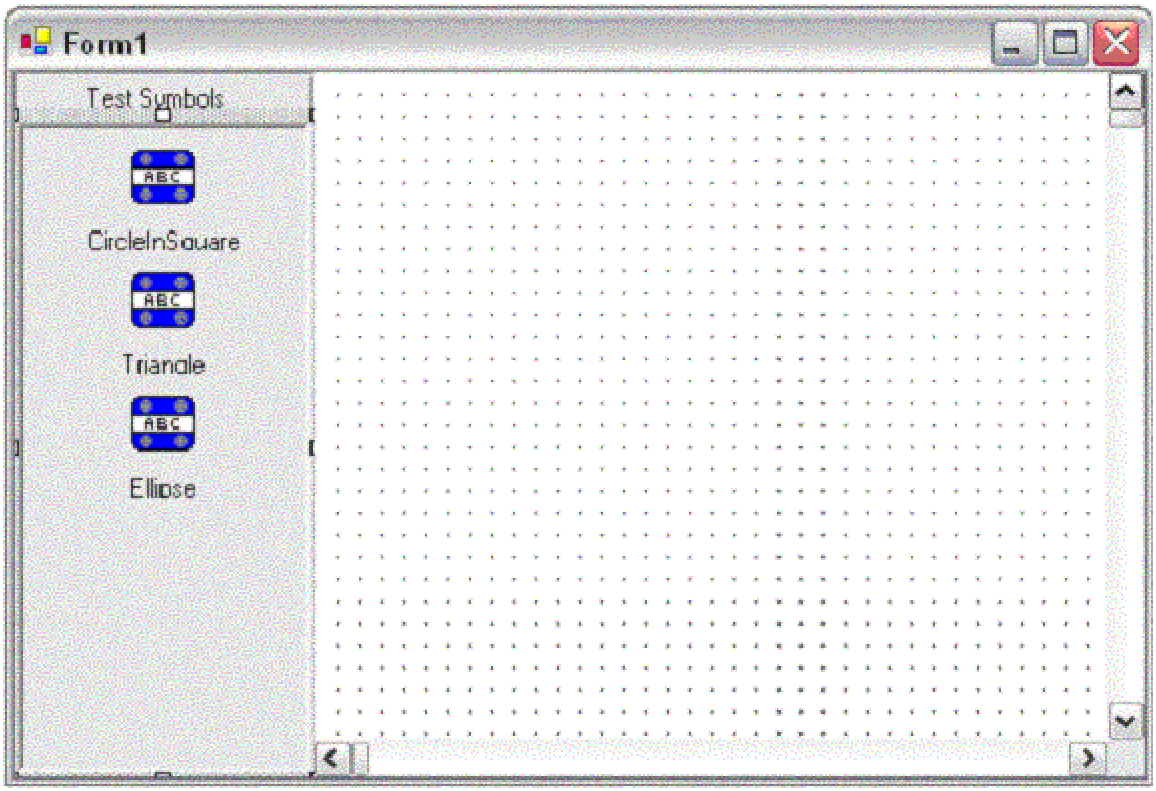

Figure 5. Symbol Palette Control.

If you compile and run it, you should be able to drag and drop the symbols from the symbol palette onto the diagram.

5.3 Add Controls to Interact with the Diagram

Next, you need to add a menu to the form so that the user can do some more interesting things with the diagram. Go to the *Windows Forms* group in the Visual Studio toolbox and add a **MainMenu** control to the form. Add two items at the top of the menu: *File* and *Tools*. Your form will now look like this.

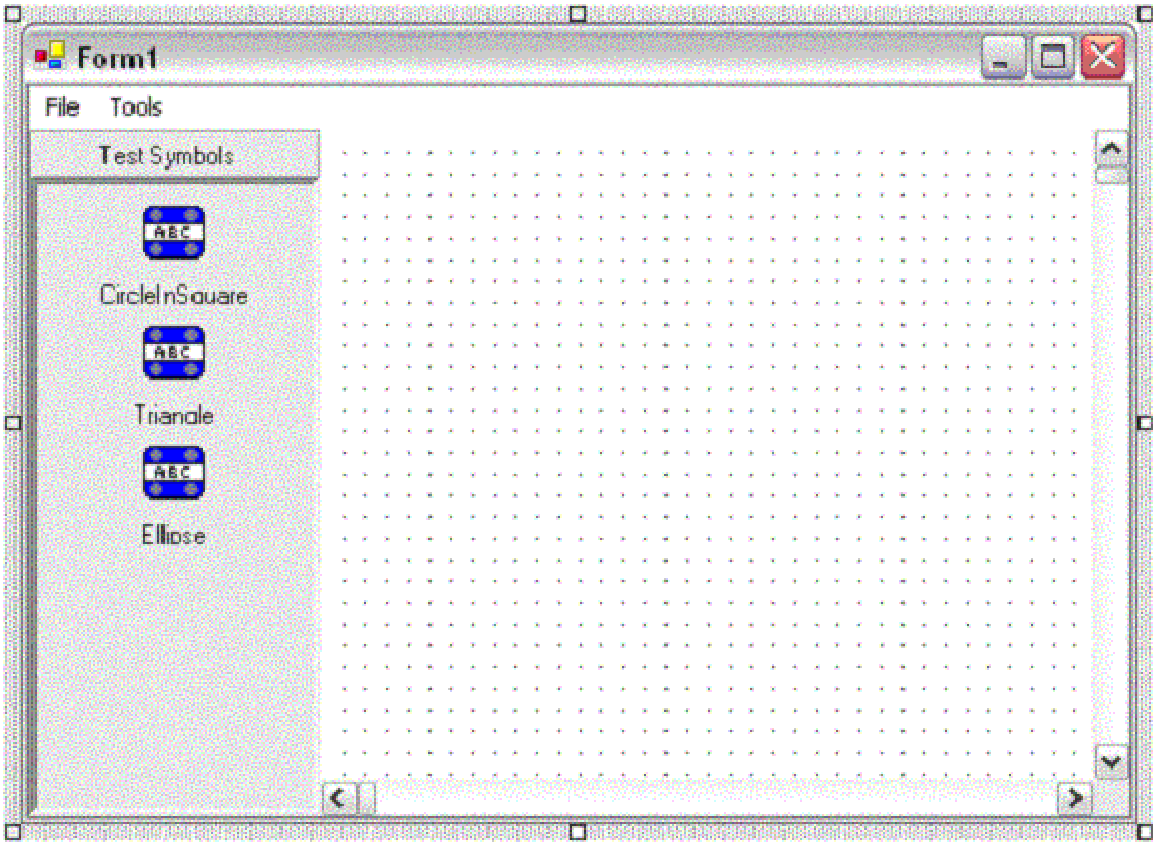

Figure 6. Figure.

Add the following items to the *File* menu.

Figure 7. Figure.

Add the following items to the *Tools* menu.

Figure 8. Figure.

5.4 Activate User-Interface Tools

Now it´s time to hook up some functionality to the menus you just added. Add the following methods to your form.

C# code

```
private void select_Click(object sender, System.EventArgs e)
{
        this.diagram1.ActivateTool("SelectTool");
}
private void pan_Click(object sender, System.EventArgs e)
{
        this.diagram1.ActivateTool("PanTool");
}
private void zoom_Click(object sender, System.EventArgs e)
{
        this.diagram1.ActivateTool("ZoomTool");
}
private void group_Click(object sender, System.EventArgs e)
{
        this.diagram1.ActivateTool("GroupTool");
}
private void ungroup_Click(object sender, System.EventArgs e)
{
        this.diagram1.ActivateTool("UngroupTool");
}
private void link_Click(object sender, System.EventArgs e)
{
        this.diagram1.ActivateTool("LinkTool");
}
```

Now select each item on the *Tools* menu and select the appropriate _Click method for the *Click* event.

Now when you run the application and select an item on the *Tools* menu, the associated user interface tool in the controller will be activated.

5.5 Add a PropertyEditor Control

Next, we´ll add a **PropertyEditor** control to the form so that the user can edit properties of objects in the diagram. Select the Essential Diagram **PropertyEditor** control in the Visual Studio toolbox and drag it onto the form. With a little formatting, your form should now look something this.

Fig 5_5

The **PropertyEditor** control is docked on the right-hand side of the form. In order to have the **PropertyEditor** automatically display the currently selected object(s) in the diagram, the Diagram control must be attached to the **PropertyEditor**. The **PropertyEditor** control has a Diagram property that can be assigned either at design-time or run-time. To set it at design-time, select the **PropertyEditor** control on the form and find the Diagram property in the *Properties* window. Clicking on the Diagram property will display a drop-down list of **Diagram** controls on the form. Select the entry *diagram1* from the drop-down list.

You can also attach the **Diagram** control to the **PropertyEditor** at run-time using the following code.

C# code

```
public Form1()
{
        //
        // Required for Windows Form Designer support
        //
        InitializeComponent();

        // Attach the diagram to the property editor
        this.propertyEditor1.Diagram = this.diagram1;
}
```

5.6 Saving and Loading the Diagram

Next, we need to add code to save the diagram and reload it. First, add an **OpenFileDialog** control and a **SaveFileDialog** control to the form from the *Windows Forms* group in the Visual Studio toolbox. The component tray of your form should look like this.

Fig 5_6

Next, add the following two functions to your form.

C# code

```
private void open_Click(object sender, System.EventArgs e)
{
        if (this.openFileDialog1.ShowDialog(this) ==
DialogResult.OK)
        {

        this.diagram1.LoadBinary(this.openFileDialog1.FileName);
        }
}

private void save_Click(object sender, System.EventArgs e)
{
        if (this.saveFileDialog1.ShowDialog(this) ==
DialogResult.OK)
        {

        this.diagram1.SaveBinary(this.openFileDialog1.FileName);
        }
}
```

Assign them the Click event of the File Open and File Save menu items, respectively. Now when you compile and run your application, you can save and reload diagrams from the File menu.

6 Using the Framework

This section explains how to use the features of the framework in your applications.

6.1 Event Model

This section explains the event model and how to handle certain events generated by the objects in a diagram. The **Model** object is the primary source of events in a diagram. It fires events when changes are made to the content of the diagram. Some examples of changes that trigger events in the **Model** are adding a new node, moving a node, and changing a property. The **View** and **Controller** classes also fire events. A complete list of events fired by the **Model, View,** and **Controller** classes can be found in the *Essential Diagram Class Reference*.

6.1.1 Event Routing

Individual nodes inside of a **Model** do not fire events. Events are routed up the node hierarchy until they reach a **Model** object, which then fires the event to its subscribers. This provides a centralized point at which event handlers can be attached. Having each node in the hierarchy would require writing code that attaches event handlers at run-time whenever nodes are added to the diagram. It would also be extremely inefficient.

Nodes in the **Model** route their events using an interface named **IDispatchNodeEvents**. When an event occurs, the node calls a virtual OnEventName callback method. For example, when a property changes in a symbol it calls its virtual OnPropertyChanged method. The callback method queries its parent for the **IDispatchNodeEvents** interface and forwards the notification onto the parent. This pattern is performed recursively until a **Model** object receives the notification through the **IDispatchNodeEvents** interface, and then it fires the event to all of its subscribers.

The following code shows the implementation of the OnPropertyChanged method in the **Symbol** class.

C# code

```
 protected virtual void OnPropertyChanged(PropertyEventArgs evtArgs)
{
        if (this.parent != null)
        {
                IDispatchNodeEvents evtDispatcher;

evtDispatcher =
   this.parent.GetService(typeof(IDispatchNodeEvents)) as
                                IDispatchNodeEvents;

                if (evtDispatcher != null)
                {
                        evtDispatcher.PropertyChanged(evtArgs);
                }
        }
}
```

Derived classes can override the OnPropertyChanged method directly in order to handle property change events. Other classes can subscribe to the PropertyChanged event exposed by the **Model** object.

The **View** object subscribes to most of the events exposed by the **Model**, so that it can update the visual representation of the diagram when changes occur. The View also exposes some events of its own, such as PropertyChanged, OriginChanged, and MagnificationChanged.

The **Controller** object fires several useful events. The ToolActivate event is fired by the controller when a user interface tool is activated. Conversely, the ToolDeactivate event is fired when a user interface tool is deactivated. The SelectionChanged event is fired by the controller when the list of selected nodes changes.

6.1.2 Example Event Handler

The following example adds an event handler to the model to receive click events on nodes. First, select the **Diagram** control on the form and then go to the Visual Studio .NET *Properties* window. Click the *Events* button in the *Properties* window to

display the events. Then find the Model property and double-click on the Click event entry. The *Properties* window should look something like this.

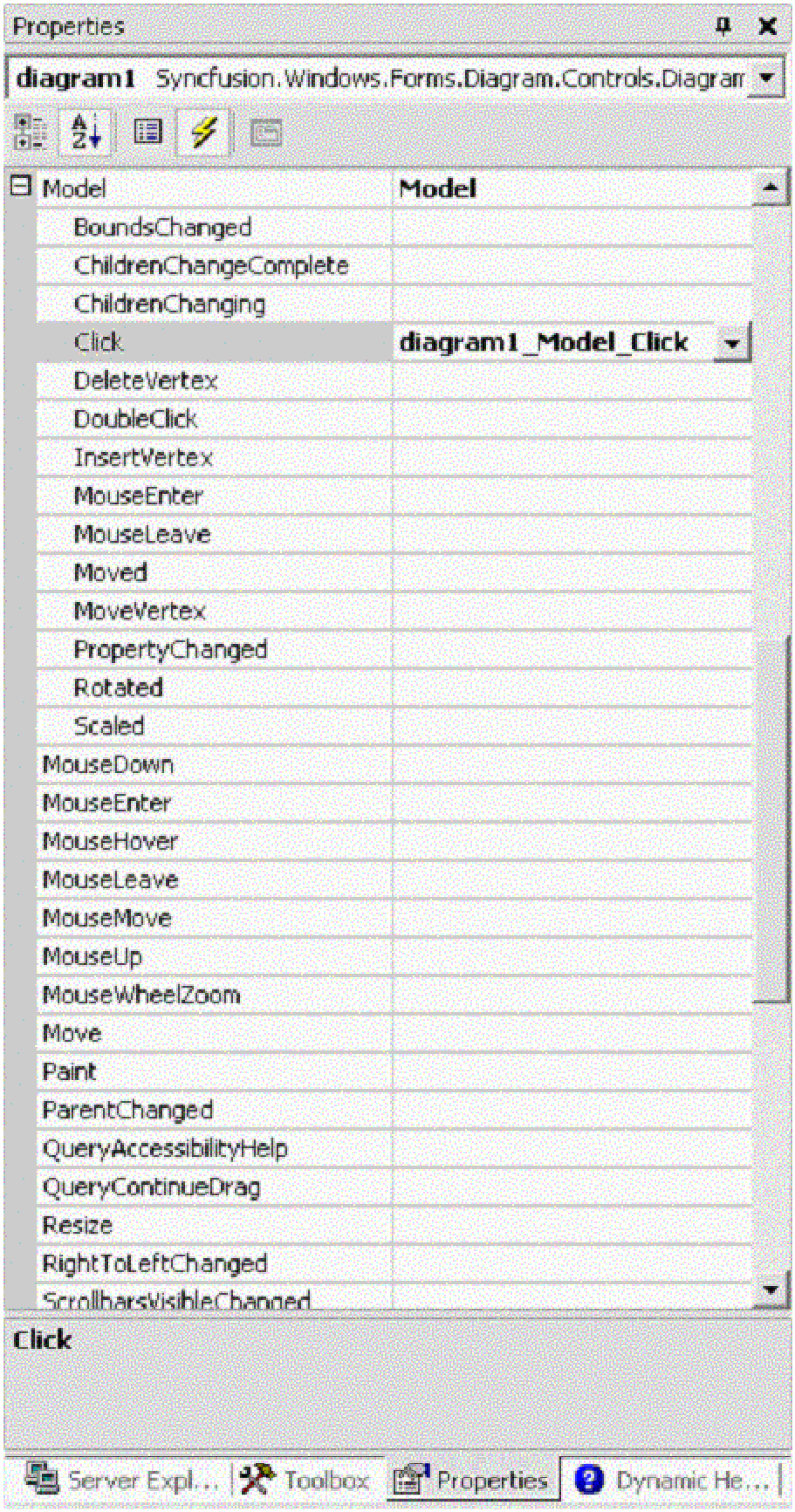

Figure 9. Figure.

Visual Studio .NET adds the diagram_Model_Click method to your form. The code below shows the diagram_Model_Click method with an implementation that pops up a message box.

C# code

```
private void diagram1_Model_Click(object sender, NodeMouseEventArgs
e)
{
        MessageBox.Show(this, e.Node.Name + " clicked");
}
```

6.2 Diagram Serialization

Serialization is the process of reading and writing a collection of objects to and from an I/O stream. Essential Diagram uses the .NET serialization framework provided by in **System.Runtime.Serialization** namespace.

The **Diagram** class contains methods for loading and saving diagrams in two of the formats supported by the .NET serialization framework. The LoadSoap and SaveSoap methods use the SOAP XML format to serialize diagrams to files and I/O streams. The LoadBinary and SaveBinary methods serialize diagrams to files and I/O streams in binary format. Using one of these method pairs is the easiest way to serialize diagrams. The code below shows how to save and load a diagram using the SOAP format.

C# code

```
private void open_Click(object sender, System.EventArgs e)
{
        if (this.openFileDialog1.ShowDialog(this) ==
DialogResult.OK)
        {

        this.diagram1.LoadSoap(this.openFileDialog1.FileName);
        }
}

private void save_Click(object sender, System.EventArgs e)
{
        if (this.saveFileDialog1.ShowDialog(this) ==
DialogResult.OK)
        {
```

```
        this.diagram1.SaveSoap(this.openFileDialog1.FileName);
        }
}
```

The load and save methods in the **Diagram** class serialize the **Model, View,** and **Controller** objects it contains. You can also serialize objects individually. All classes that can be serialized are marked with the **SerializableAttribute** and implement the **ISerializable** interface. The following code shows how to serialize a **Model** object to a stream in binary format.

C# code

```
public virtual void SaveModel(Model mdl, Stream strmOut)
{
        BinaryFormatter formatter = new BinaryFormatter();
        formatter.Serialize(strmOut, mdl);
}

public virtual Model LoadModel(Stream strmIn)
{
        BinaryFormatter formatter = new BinaryFormatter();
        return (Model) formatter.Deserialize(strmIn);
}
```

6.3 Symbol Palette Serialization

A symbol palette is a derived **Model** that contains a collection of **SymbolModel** objects. Symbol palettes can be created using the Symbol Designer utility, which is the topic of a later section. Typically, symbol palettes are created using the symbol designer and loaded into your application at run-time. There are a couple of different ways to load a symbol palette into your application.

Symbol palettes are marked with the **SerializableAttribute** and implement the **ISerializable** interface, and therefore can be serialized using the .NET serialization framework. The following code demonstrates loading a symbol palette using the SOAP formatter.

C# code

```
 public SymbolPalette LoadPalette(string fileName)
{
        SymbolPalette pal = null;
        FileStream iStream = new FileStream(fileName,
FileMode.Open);
        SoapFormatter formatter = new SoapFormatter();
        pal = (SymbolPalette) formatter.Deserialize(iStream);
        iStream.Close();
        return pal;
```

```
}
```

The corresponding code to save palettes is shown below.

C# code

```
public void SavePalette(SymbolPalette pal, string fileName)
{
        FileStream oStream = new FileStream(fileName,
FileMode.Create);
        SoapFormatter formatter = new SoapFormatter();
        formatter.Serialize(oStream, pal);
        oStream.Close();
}
```

The **PaletteGroupView** control provides an easy way to serialize a symbol palette to and from the resource file of a form. At design-time, you can attach a symbol palette to a **PaletteGroupView** control on a form. Select the **PaletteGroupView** and then click on the Palette property in the Visual Studio .NET *Properties* window. This will open a standard file open dialog and allow you to select a symbol palette file that has been created with the Symbol Designer utility. The result is that the symbol palette is serialized to the resource file of the form. The following line of code is automatically added to the form´s InitializeComponent method in order to deserialize the symbol palette from the resource file at run-time.

C# code

```
private void InitializeComponent()
{
  // . . . other initialization . . .

this.paletteGroupView1.LoadPalette(typeof(MyApp.Form1).Assembly,
                                   "MyApp.Form1",
                                   "paletteGroupView1.Palette");
}
```

If you´re not using the **PaletteGroupView** control in your application, you can still compile your symbol palettes in a resource file. The following method loads symbol palettes from a resource file.

C# code

```
public SymbolPalette LoadPalette(string resName)
{
        SymbolPalette palette = null;
        string bsNm;
        System.Reflection.Assembly asm;
```

```
        System.Resources.ResourceManager resMgr;

        ////////////////////////////////////////
        // Change the next two lines of code to
        // reference the class name of your form
        asm = typeof(MyApp.Form1).Assembly;
        bsNm = "MyApp.Form1";
        ////////////////////////////////////////

// Create a resource manager for the resource file
resMgr = new System.Resources.ResourceManager(bsNm, asm);

// Read raw symbol palette from the resource file as
// an array of bytes
        object resObj = resMgr.GetObject(resName);
        if (resObj != null && resObj.GetType() == typeof(byte[]))
        {
                // Load the byte array into a memory stream so that
it
                // can be serialized
                System.IO.MemoryStream strmRes;
strmRes = new MemoryStream((byte[]) resObj);
// Create a binary formatter and deserialize the stream
                BinaryFormatter formatter = new BinaryFormatter();
palette = (SymbolPalette) formatter.Deserialize(strmRes);
                strmRes.Close();
        }
        return palette;
}
```

You can include the method above into your own forms in order to load symbol palettes from the form´s resource file. You will have to change the values assigned to the **asm** and **bsNm** variables to match the name of your form class.

6.4 Graph Navigation

The **Model** object can be viewed as a graph of nodes and edges, where symbols are nodes and links are edges. Two interfaces are defined to help facilitate navigating the **Model** as a graph. The first is **IGraphNode**, which is implemented by the **Symbol** class. The second is **IGraphEdge**, which is implemented by the **Link** class.

6.5 Subclassing Symbols

New classes can be derived from the **Symbol** class in order to implement application specified behavior. Derived symbol classes encapsulate data and behavior that can be easily reused in your applications. They can programmatically load the contents of the symbol, add custom data and methods, and change dynamically in response to events.

The listing below shows a derived symbol class called **MySymbol** that loads its contents programmatically and responds to mouse events. **MySymbol** objects

consist of an ellipse inside of a rectangle. When the mouse moves over a **MySymbol** object, the color of the rectangle changes from Khaki to Green. When a **MySymbol** object is clicked, the color of the ellipse changes. This sample can be found in the Samples directory under DynamicSymbol.

C# code

```
 using System;
using System.Drawing;
using Syncfusion.Windows.Forms.Diagram;

namespace DynamicSymbol
{
        public class MySymbol : Symbol
        {
                private Syncfusion.Windows.Forms.Diagram.Rectangle
outerRect = null;
                private Ellipse innerEllipse = null;
                private int curEllipseColor = 0;

                static System.Drawing.Color[] ellipseColors =
                {
                        Color.LightBlue,
                        Color.Silver,
                        Color.Yellow,
                        Color.MidnightBlue
                };

                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.FillStyle.Color = Color.Khaki;
                        this.AppendChild(outerRect);

                        // Add an inner ellipse
                        this.innerEllipse = new Ellipse(10, 10, 100,
60);
                        this.innerEllipse.FillStyle.Color =
                                ellipseColors[this.curEllipseColor];
                        this.AppendChild(innerEllipse);
                }

                protected override void
OnMouseEnter(NodeMouseEventArgs evtArgs)
                {
                        this.outerRect.FillStyle.Color = Color.Green;
                }
```

```
                protected override void
OnMouseLeave(NodeMouseEventArgs evtArgs)
                {
                        this.outerRect.FillStyle.Color = Color.Khaki;
                }

                protected override void OnClick(NodeMouseEventArgs
evtArgs)
                {
                        if (this.curEllipseColor ==
ellipseColors.Length-1)
                        {
                                this.curEllipseColor = 0;
                        }
                        else
                        {
                                this.curEllipseColor++;
                        }
                        this.innerEllipse.FillStyle.Color =
ellipseColors[this.curEllipseColor];
                }
        }
}
```

6.6 Sub-classing the Diagram Control

New classes can be derived from the **Diagram** control class in order to implement application specific functionality. The most common reason for sub-classing the **Diagram** control is to plug-in alternate implementations of the model, view, or controller objects. The listing below shows a derived **Diagram** class that overrides the creation of the model, view, and controller objects.

C# code

```
using System;
using Syncfusion.Windows.Forms.Diagram;
using Syncfusion.Windows.Forms.Diagram.Controls;

namespace Tutorial
{
        public class MyDiagram : Diagram
        {
                public MyDiagram()
                {
                }

                public override Model CreateModel()
                {
                        return new MyModel();
                }
```

```
                public override View CreateView()
                {
                        return new MyView();
                }

                public override Controller CreateController()
                {
                        return new MyController();
                }

                protected class MyModel : Model
                {
                }

                protected class MyView : View
                {
                }

                protected class MyController : Controller
                {
                }
        }
}
```

6.7 Logical Units

Objects in the model store points and sizes in world coordinates. The unit of measure for world coordinates depends on the Model.MeasurementUnits property. By default, the unit of measure is Pixel. That means that each logical unit in the model maps to a single pixel on the output device.

Using pixels as the unit of measure is fine for applications that do not require real-world measurements and can tolerate some amount of device dependence. When using pixels as the unit of measure, remember that the actual size of a pixel depends on the resolution of the output device. A common resolution for computer screens is 96 DPI (dots per inch), which means that each pixel is 1/96 of an inch. On a printer that is 300 DPI, a pixel is 1/300 of an inch. Drawing the same 10x10 rectangle on both of these devices will yield different results.

The Model.MeasurementUnits property can be to use real-world units of measurement such as inches or millimeters. The possible values come from the **System.Drawing.GraphicsUnit** enumeration and are listed below.

GraphicsUnit value	**Description**
Pixel	Each logical unit is 1 pixel on the output device
Inch	Each logical unit is 1 inch
Millimeter	Each logical unit is 1 millimeter
Point	Each logical unit 1/72 of an inch
Display	Each logical unit is 1/75 of an inch
Document	Each logical unit is 1/300 of an inch

If you set the Model.MeasurementUnits property to Inch and draw a 1x1 rectangle, it should be roughly 1 inch by 1 inch on any output device, regardless of the device´s resolution.

When you change the the Model.MeasurementUnits property, values stored in the model are automatically recalculated to match the new unit of measure. This prevents things from resizing when you change your logical unit of measure. For example, if your model uses pixel units and contains a rectangle that is 96x96 units and you then change the model to use inches, the rectangle will be converted to be 1x1 logical units (assuming your screen is 96 DPI).

The **Syncfusion.Windows.Forms.Diagram.Measurements** class contains several static methods for converting values between different units of measure.

6.8 Printing

The **DiagramPrintDocument** class implements printing for diagram models. It derives from the **System.Drawing.Print.PrintDocument** class provided by the .NET printing framework. The **DiagramPrintDocument** class overrides the OnPrintPage method and provides an implementation that prints the requested page of the diagram. The OnPrintPage method is called by the .NET printing framework after a print job is started.

The **Diagram** control provides a method called CreatePrintDocument that creates and initializes a **DiagramPrintDocument**. To print a diagram, all your application has to do is call the CreatePrintDocument method, and then call the Print method on the **DiagramPrintDocument** that is returned. The CreatePrintDocument method automatically initializes the default page settings in the print document using the page settings stored in the **Model** object.

The following code creates a print document, prompts the user with the standard print dialog allowing them to change the default page settings, and then prints the diagram.

C# code

```
 public void PrintDiagram(Diagram diagram)
{
        PrintDocument printDoc = diagram.CreatePrintDocument();
        PrintDialog printDlg = new PrintDialog();
        printDlg.Document = printDoc;
        if (printDlg.ShowDialog(this) == DialogResult.OK)
        {
                printDoc.Print();
        }
}
```

Page settings are stored in the **Model** class and can be edited using the **PageSetupDialog** provided by .NET. The following code edits the page settings in the model.

C# code

```
 private void EditPageSettings(Diagram diagram)
{
        PageSetupDialog dlg = new PageSetupDialog();
        dlg.PageSettings = diagram.Model.PageSettings;
        if (dlg.ShowDialog(this) == DialogResult.OK)
        {
                diagram.Model.PageSettings = dlg.PageSettings;
        }
}
```

7 Using the Symbol Designer

The Symbol Designer utility allows you to create symbol palettes that can be used in your diagramming applications. A symbol palette is a collection of symbol models that can be used to create symbols at run-time. A symbol model contains all of the information needed to construct a symbol. Using the Symbol Designer, you can add shapes, text, bitmap images, ports, and labels to the symbol models in a symbol palette.

7.1 Where to find the Symbol Designer

All of the source code for the Symbol Designer is included with Essential Diagram and can be found in the [InstallDir]/Essential Suite/Diagram/Symbol Designer directory. The Symbol Designer is built on the Essential Diagram framework and uses the **Diagram** control, so it serves as a dual role as both a utility and a sample. A Visual Studio solution is included so that you can re-build the Symbol Designer, and it includes both Debug and Release configurations.

A release build of the Symbol Designer is shipped with Essential Diagram, and a shortcut to it is added to the Windows Start menu under Syncfusion > Essential Suite > Diagram.

7.2 User Interface Overview

The Symbol Designer is an MDI (Multiple Document Interface) application. Only one symbol palette can be opened at a time, but multiple symbol models in that symbol palette can be open simultaneously. The Symbol Designer user interface is shown below.

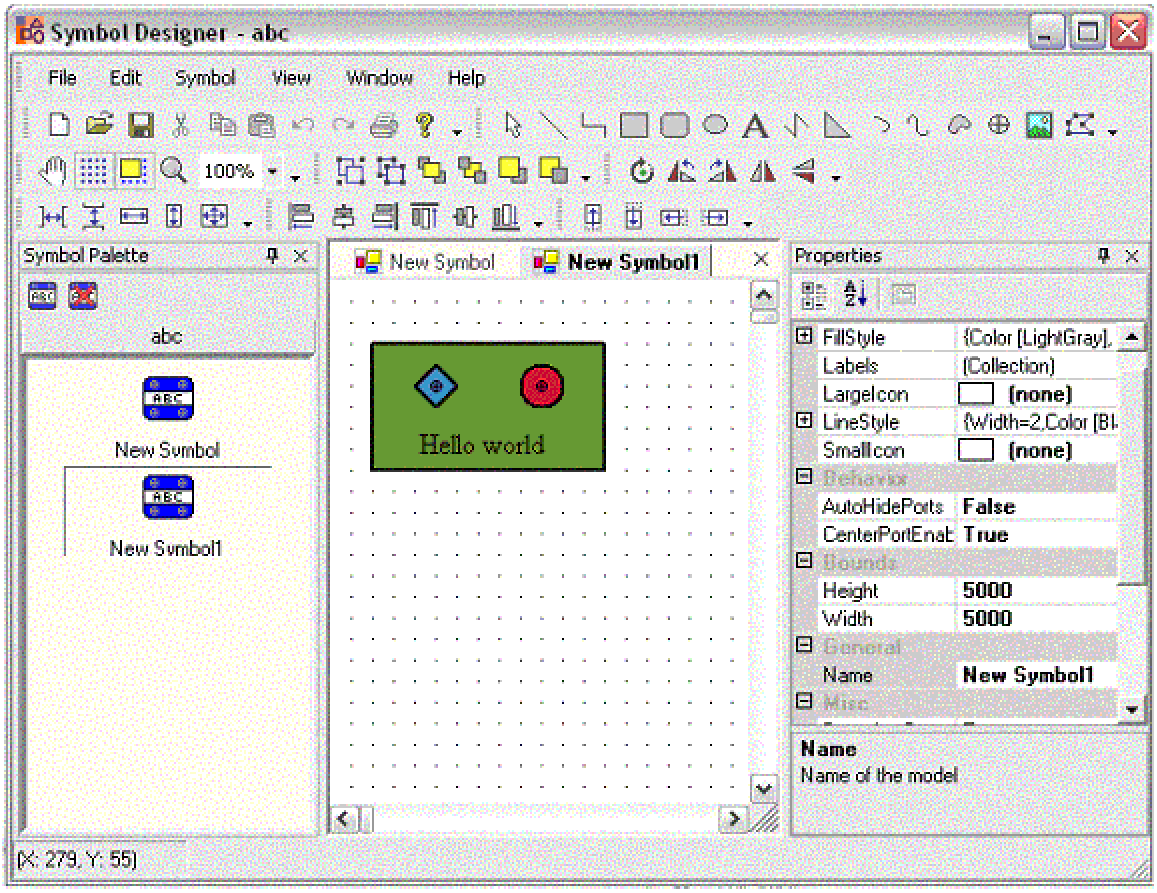

Figure 10. Figure.

The screen shot above shows the Symbol Designer editing a symbol palette named "abc" that contains two symbol models - one named "New Symbol" and one named "New Symbol1".

Symbol Palette

The *Symbol Palette* window is docked on the left-hand side of the main frame window. It contains a **PaletteGroupBar** control which displays a list of icons for the symbol models in the symbol palette. The name of the symbol palette is displayed on a group bar item above the icons - in this case "abc". Since only one symbol palette can be open in the Symbol Designer at a time, the **PaletteGroupBar** will contain only one group bar item at any given time.

MDI Child Windows

There are two MDI child windows open, which are used for editing the symbol models in the palette. The Symbol Designer supports a tabbed MDI interface, which displays MDI child windows on tabs in the client area of the main frame window. The tabbed MDI interface can be disabled from the *Window* menu, which will cause a traditional

MDI interface to be used instead. The *Window* menu also contains commands for reorganizing and closing MDI child windows.

Properties

The *Properties* window is docked on the right-hand side of the main frame. It contains a **PropertyEditor** control that displays the properties of the currently selected object. In the screen shot above, "New Symbol1" is the currently selected symbol model and its properties are displayed in the *Properties* window.

Toolbars

The Symbol Designer contains a set of toolbars that are used to activate user interface tools for editing symbol models and perform various other tasks. The toolbars are explained in more detail later in this document.

Status Bar

The status bar at the bottom of the main frame window displays the current position of the mouse in the diagram.

Options Window

The *Options* window is used to set properties of the diagram that are not available through the *Properties* window. A screen shot of the *Options* window is shown below.

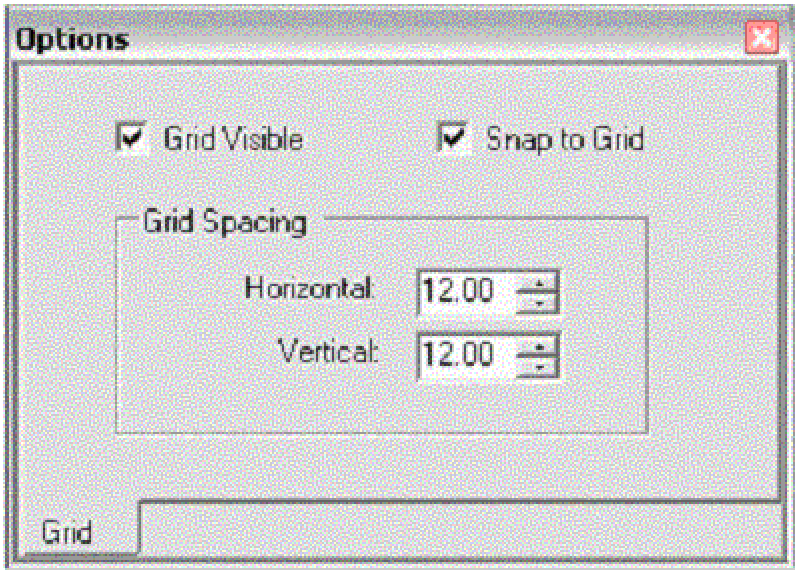

Figure 11. Figure.

Hiding and Showing Windows

The *View* menu can be used to show and hide the windows docked in the main frame window. The *View* menu is shown below.

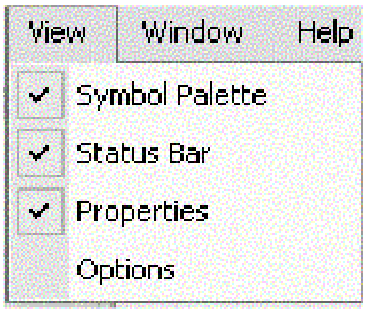

Figure 12. Figure.

Docking Windows

The Symbol Designer uses the Syncfusion Tools **DockingManager** to allow the windows in the main frame to be docked and moved around by the user. The *Symbol Palette* and *Properties* windows support docking, and all of the toolbars can be docked and customized.

7.3 Creating a new Symbol Palette

Launch the Symbol Designer from the Syncfusion > Essential Suite > Diagram entry in the Windows Start menu. Now that you have the Symbol Designer running, the first thing you need to do is to create a new Symbol Palette. You can do this either by selecting File|New from the menu or clicking on the New Palette button in the Standard toolbar.

You will be prompted to enter a name for the new symbol palette. The name you enter here will be displayed on the group bar of the **PaletteGroupBar** control. After you enter the palette name and click OK, a new symbol palette is created and attached to the **PaletteGroupBar** control. It will look like this.

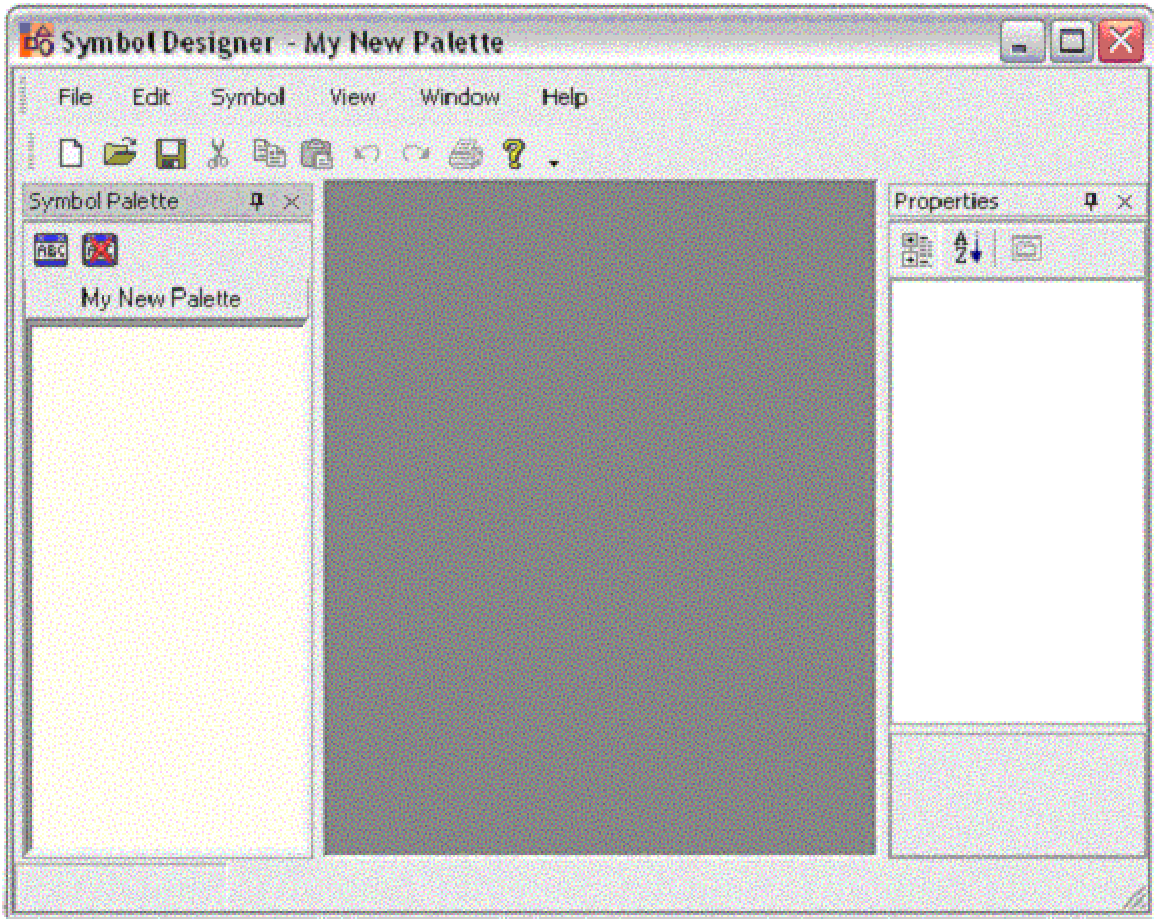

Figure 13. Figure.

7.4 Adding Symbols to a Palette

Now that you have a symbol palette, you can start adding new symbols to it. To add a new symbol, select the Symbol|Add menu item. The icon looks like this.

Figure 14. Figure.

You can also find it on a toolbar button above the **PaletteGroupBar** control.

After clicking Symbol|Add, the Symbol Designer creates a new symbol and gives it a default name of "New Symbol". The new symbol is loaded into a diagram inside of an MDI child window. Now you can start editing your new symbol.

7.5 Symbols versus Symbol Models

When using the Symbol Designer, it is important to keep in mind the distinction between symbols and symbol models. The Symbol Designer edits symbol models, not symbols. Symbol models are used to create symbols at run-time. They are similar in the sense that they both contain the same child nodes and properties, but the symbol model is only used at design-time. A symbol model is the design-time representation of a symbol. The Symbol Designer occasionally uses the terms "symbol model" and "symbol" interchangeably. Since the Symbol Designer is a design-time utility, it only deals with the design-time representation of a symbol which is the symbol model.

7.6 Editing Symbol Models with the Symbol Designer

Now that you have created a symbol model in a palette, it´s time to add some content. The Symbol Designer uses the **Diagram** control for editing symbol models, so the user interface is very similar to the user interface of your own diagramming applications. The remainder of this section gives a brief description of the user interface tools provided by the Symbol Designer for editing symbol models.

7.6.1 Drawing Tools

The Drawing toolbar contains tools for drawing and editing shapes, and adding text, ports, and bitmap images. The Drawing toolbar is shown below, followed by a brief description of each tool.

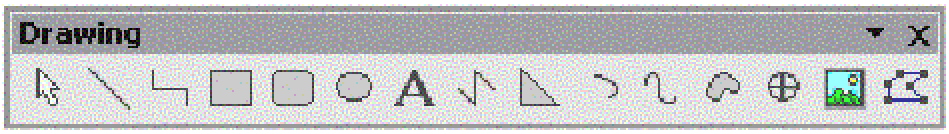

Figure 15. Figure.

- *Select Tool* - Activates the select tool and deactivates any other active tools . This is the default tool. Objects that are selected using the Select Tool are drawn with select handles. Multiple objects can be selected using the select tool by holding down the Shift key or by holding down the left mouse button and dragging. The last object select is the anchor object, which is used as a reference point for some tools such as the alignment tools.
- *Line Tool* - Used to draw a line with two points.

- *Orthogonal Line Tool* - Used to draw a line between two points using right angle bends
- *Rectangle Tool* - Used to draw a rectangle shape
- *Rounded Rectangle Tool* - Used to draw a rectangle with rounded corners
- *Ellipse Tool* - Used to draw ellipses
- *Text Tool* - Used to add text nodes to the symbol
- *Poly Line Tool* - Used to add lines containing two or more points to the symbol
- *Polygon Tool* - Used to add filled polygons to the symbol
- *Arc Tool* - Used to add arcs to the symbol
- *Curve Tool* - Used to add open curves to the symbol
- *Closed Curve Tool* - Used to add closed curves to the symbol
- *Port Tool* - Used to add circle ports to the symbol
- *Image Tool* - Used to add bitmap images to the symbol
- *Vertex Edit Tool* - Used to edit the vertices of existing shapes in the symbol

7.6.2 Rotate Tools

The Rotate toolbar contains tools for rotating objects in the symbol model. The Rotate toolbar is shown below, followed by a brief description of each tool.

Figure 16. Figure.

- *Rotate Tool* - Activates the rotate tool. To rotate, move the mouse over the object you want to rotate and then hold down the left mouse button. Move the mouse to rotate the object and then release the left mouse button to complete the rotation.
- *Rotate Left* - Rotates the currently selected objects left by 90 degrees.
- *Rotate Right* - Rotates the currently selected objects right by 90 degrees.
- *Flip Vertical* - Rotates the currently selected objects by 180 degrees around their local Y axis.
- *Flip Horizontal* - Rotates the currently selected objects by 180 degrees around their local X axis.

7.6.3 Node Tools

The Node toolbar contains tools for grouping objects and changing their Z-order. The Node toolbar is shown below, followed by a brief description of each tool.

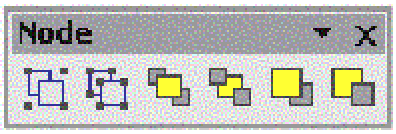

Figure 17. Figure.

- *Group Tool* - Creates a new group using the currently selected objects.
- *Ungroup Tool* - Destroys the currently selected group.
- *Bring to Front* - Moves the currently selected object to the top of the Z-order of the layer it belongs to.
- *Send to Back* - Moves the currently selected object to the bottom of the Z-order of the layer it belongs to.
- *Bring Forward* - Moves the currently selected object up by 1 in the Z-order of the layer it belongs to.
- *Send Backward* - Moves the currently selected object down by 1 in the Z-order of the layer it belongs to.

7.6.4 Align Tools

The Align toolbar contains tools for aligning selected objects relative to the anchor object. The anchor object is the last object selected by the Select tool. The Align toolbar is shown below, followed by a brief description of each tool.

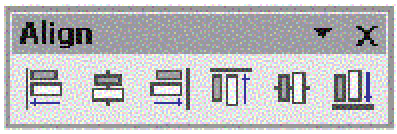

Figure 18. Figure.

- *Align Left* - Moves the selected objects so that their left boundary lines up with the left boundary of the anchor object.
- *Align Center* - Moves the selected objects along the X axis so that their centers line up with center point of the anchor object.
- *Align Right* - Moves the selected objects so that their right boundary lines up with the right boundary of the anchor object. object to the top of the Z-order of the layer it belongs to.
- *Align Top* -Moves the selected objects so that their top boundary lines up with the top boundary of the anchor object.
- *Align Middle* - Moves the selected objects along the Y axis so that their centers line up with the center point of the anchor object.

- *Align Bottom* - Moves the selected objects so their bottom boundary lines up with the bottom boundary of the anchor object.

7.6.5 Layout Tools

The Layout toolbar contains tools for spacing and sizing objects in the symbol model. The Layout toolbar is shown below, followed by a brief description of each tool.

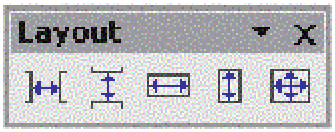

Figure 19. Figure.

- *Space Across* - Evenly spaces the selected objects along the X axis.
- *Space Down* - Evenly spaces the selected objects along the Y axis.
- *Same Width* - Sets the width of the selected objects to match the width of the anchor object.
- *Same Height* - Sets the height of the selected objects to match the height of the anchor object.
- *Same Size* - Sets the width and height of the selected objects to match the anchor object.

7.6.6 Nudge Tools

The Nudge toolbar contains tools for moving objects in the symbol model. The Nudge toolbar is shown below, followed by a brief description of each tool.

Figure 20. Figure.

- *Nudge Up* - Moves the selected objects up by 1 grid unit.
- *Nudge Down* - Moves the selected objects down by 1 grid unit.
- *Nudge Left* - Moves the selected objects left by 1 grid unit.
- *Nudge Right* - Moves the selected objects right by 1 grid unit.

7.6.7 View Tools

The View toolbar contains tools for controlling the view of the current symbol model diagram. The View toolbar is shown below, followed by a brief description of each tool.

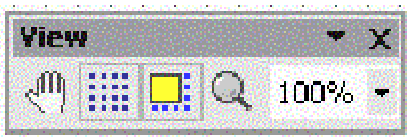

Figure 21. Figure.

- *Pan Tool* - Activates the pan tool. Allows the user to click and drag to scroll the diagram in any direction.
- *Show Grid* - Toggles the visibility of the grid.
- *Snap to Grid* - Toggles the snap to grid feature.
- *Zoom Tool* - Activates the zoom tool. Allows the user to left click to zoom in and right click to zoom out.
- *Magnification* - Displays the current magnification percentage.

7.7 Using the Property Editor

The *Properties* window contains a **PropertyEditor** control that displays the properties of the currently selected object in the diagram. It also allows you to edit the properties. You can have many different symbol models (diagrams) open at any given time, each of which contains many objects. If you´re not sure which object is currently displayed in the *Properties* window, look for the Name property.

When you first open a symbol model for editing, the properties displayed in the *Properties* window belong to the **SymbolModel** object. If no other object in the diagram is selected, the properties of the symbol model are displayed in the *Properties* window. When you want to edit the properties of the symbol model, all you need to do is click in the **Diagram** control at a location that doesn´t contain any objects. This will clear the selection list and load the properties of the **SymbolModel** object into the *Properties* window.

7.8 Saving and Loading Symbol Palettes

The Symbol Designer utility saves symbol palettes to files that have a default extension of .edp (Essential Diagram Palette). The File|Save and File|Open commands can be accessed from either the File menu or the Standard toolbar. Clicking on File|Save and File|Open will open a standard file save/open dialog that prompts you to select a filename.

Symbol palettes created with the Symbol Designer utility can be loaded into your application at run-time in a variety of ways. The Symbol Designer utility serializes symbol palettes using the SOAP format, which is a text-based XML format. Read the section titled *Symbol Palette Serialization* for details about how symbol palettes are serialized.

7.9 Symbol Size

The bounds of the symbol model are set to a default size. You can change the size of the symbol model by selecting it, going to the *Properties* window and setting the Width and Height properties in the Bounds category. This changes the size of the canvas on which editing occurs, but it does not affect the actual size of symbols created by the symbol model. The size of symbols created by a symbol model is determined by the aggregate bounds of the objects it contains.

The screen shot below shows a symbol model with a model size of 220x250. The red rectangle outlines the actual bounds of symbols created from this model.

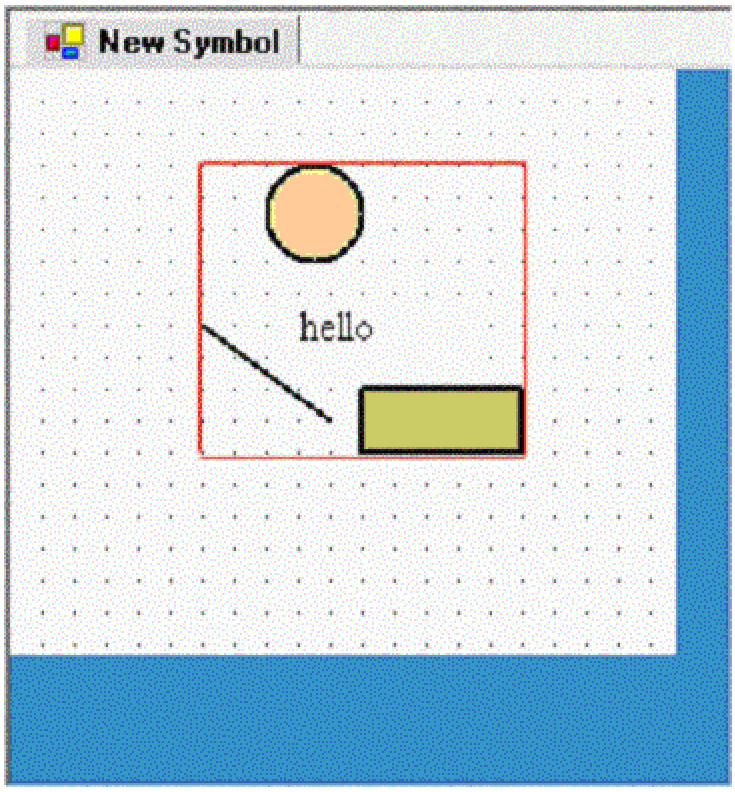

Figure 22. Figure.

7.10 Adding Labels

Labels can be added to the symbol model through the Labels property in the *Properties* window. Select the symbol model into the *Properties* windows by clicking on the diagram. Then click on the Labels property. This will open a collection editor that will allow you to add new labels and edit existing labels. The collection editor is shown below.

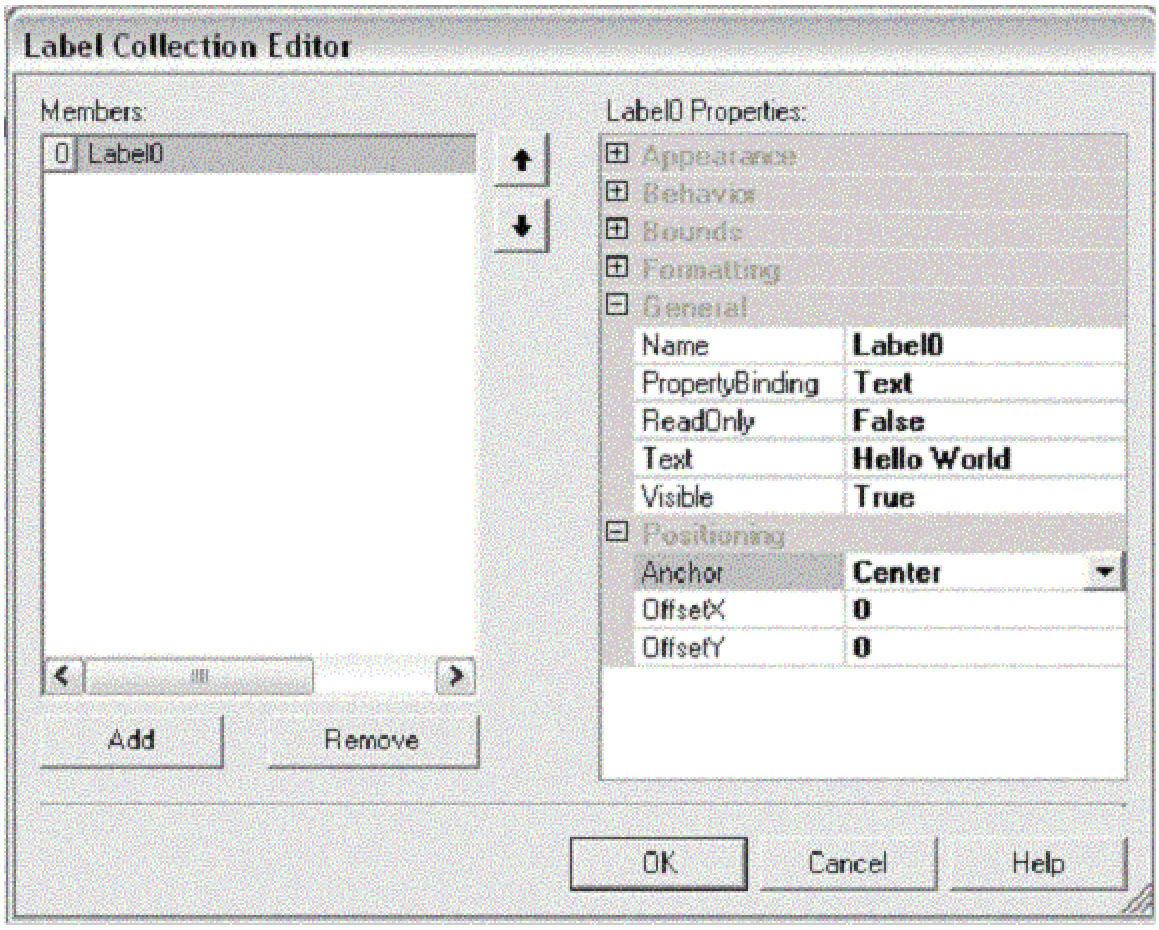

Figure 23. Figure.

Note that some of the property categories in the screen shot above have been collapsed.

The PropertyBinding property determines where the text value of the label comes from. The label can be bound to either the "Text" property or it can be bound to the "ContainerName". The "Text" property is an independent string value maintained by the label. The "ContainerName" property maps to the "Name" property of the symbol

that owns the label. This is handy when you want to add a label to a symbol that always displays the name of the symbol.

The Anchor property determines the location at which the label is attached to the symbol. It can be set to any control point on the symbol´s bounding box. The default value for Anchor is "Center". The OffsetX and OffsetY properties allow you to move the label relative to the anchor point.

7.11 Adding Ports

Ports can be added to the symbol model using the **PortTool** on the *Drawing* toolbar. The **PortTool** button looks like this.

Figure 24. Figure.

When you click it, the cursor will change to a crosshair. The next left mouse button click will drop a new **CirclePort** onto the symbol model. Keep in mind that all symbols have an implicit **CenterPort** located at the center of the symbol. The center port can be enabled and disabled by setting the CenterPortEnabled property on the symbol model. The AutoHidePorts property of the symbol model controls the visibility of ports when the user-interface is not activating creating connections. Setting the AutoHidePorts property to true will cause ports to be displayed only when the user is linking symbols together.

7.12 Binding to a Sub-classed Symbol

New classes can be derived from the **Symbol** class in order to provide functionality specific to your application. In order to use your derived symbol classes, the **SymbolModel** must be told what type of symbol to instantiate when it is asked to create symbols. The PlugInAssembly and PlugInClass properties of the **SymbolModel** allow you to specify the type of symbol object instantiated by a symbol model. The default value of the PlugInAssembly property is **Syncfusion**. Diagram. The default value of the PlugInClass property is **Syncfusion.Windows.Forms.Diagram.Symbol**. The PlugInClass property must specify a class that is derived from the **Symbol** class.