



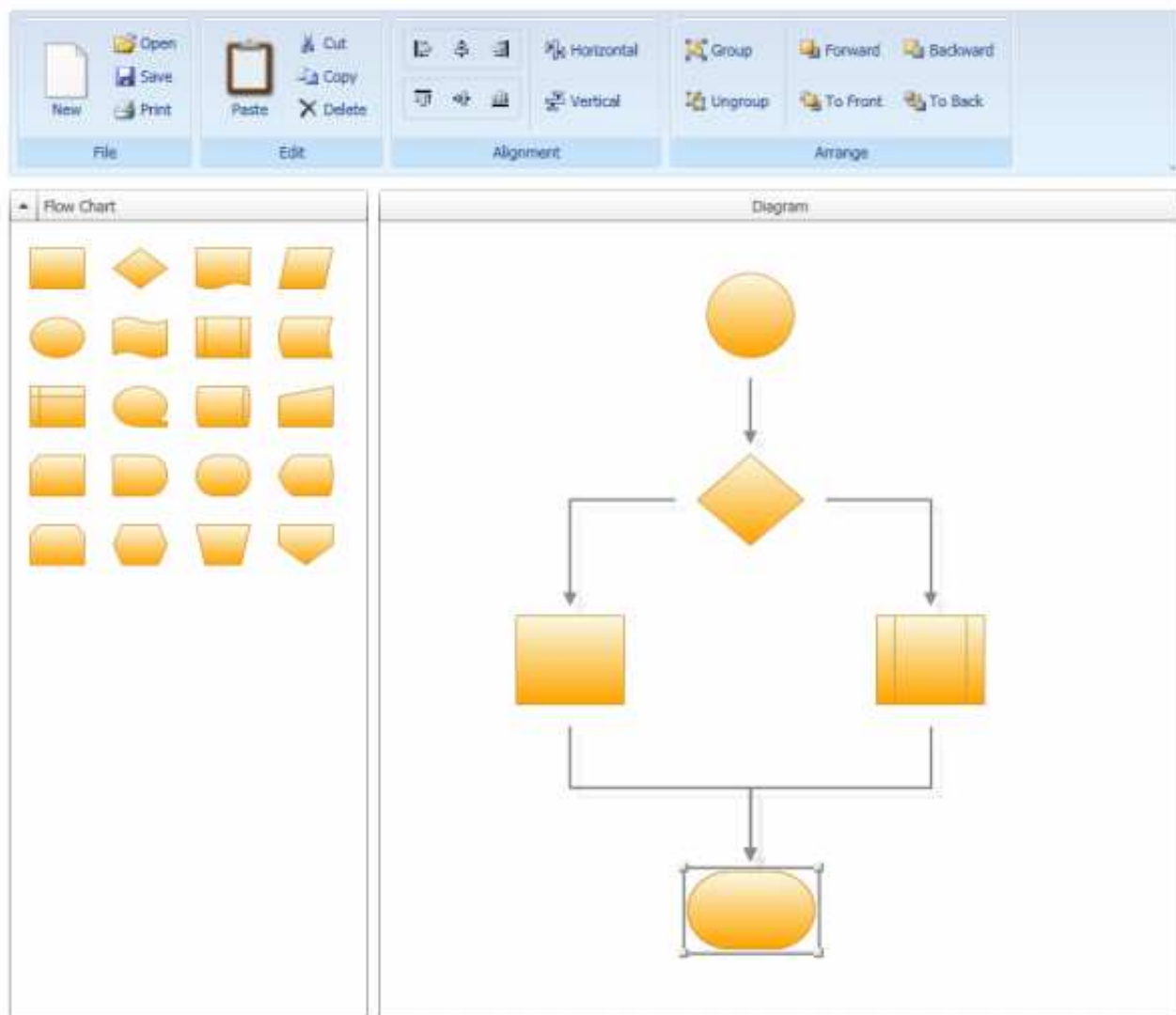
Articles » Platforms, Frameworks & Libraries » Windows Presentation Foundation » General

WPF Diagram Designer - Part 4

By **sukram**, 26 Mar 2008

★★★★★ 4.95 (165 votes)

[Download source - 72.27 KB](#)



- [Part 1](#) - Drag, resize and rotate items on a canvas
- [Part 2](#) - Toolbox, drag & drop, rubberband selection
- [Part 3](#) - Connecting items

Introduction

In this article, I have added the following commands:

- **Open, Save**
- **Cut, Copy, Paste, Delete**
- **Print**
- **Group, Ungroup**
- **Align** (Left, Right, Top, Bottom, Centered horizontal, Centered vertical)
- **Distribute** (horizontal, vertical)
- **Order** (Bring forward, Bring to top, Send backward, Send to back)

Note: I will only support Visual Studio 8.0 on .NET 3.5 !

Commands

The way I use WPF commands is straight forward, as described in the WPF SDK documentation, no extra infrastructure.

Grouping

My first approach to group items was to use a **DesignerItem** object that should work as a group container. For this, I created a new instance of the **DesignerItem** class with a **Canvas** object as its content. On this canvas, I planned to position the designer items to be grouped. But before I could put the items on the group canvas, I had to remove them from the designer canvas because in WPF an element cannot be a child of two elements. If you try, you will get an **InvalidOperationException** with the following message:

```
"Specified element is already the logical child of another element.  
Disconnect it first."
```

So I removed the items from the designer canvas and put them on the group canvas. Now it is interesting to understand what WPF did behind the scenes: as soon as I removed an item from the designer canvas, its template was unloaded and when I added it to the group canvas, a new template was loaded. Now do you remember the last article where I showed you how to connect designer items? There I connected items via connectors, connectors that were part of the designer item's template, a template that is lost as soon as I remove the item from the designer canvas. You see the problem? I have connected designer items via their templates and so the designer item itself has absolutely no information about existing connections. All connection related information is isolated in the designer item's template.

Imagine a database diagram where the designer item's content is a database table. The table would never recognize any relation to other tables. One solution would be to tunnel the information from the template to the designer item to the table. A better solution is to redesign the application and divide the whole bulk into separate parts, e.g.

- Template (view)
- Designer item (view model)
- Database table (model)

I will not start redesigning this code in the midst of an article, instead I will ride this 'view-only-approach'

until the end of this article. The more painful this ride is, the more welcome a better solution will be. (I will cover a model backed designer in a future article.)

So let's continue. An alternative approach to group designer items uses the following interface:

```
public interface IGroupable
{
    Guid ID { get; }
    Guid ParentID { get; set; }
    bool IsGroup { get; set; }
}
```

The idea is that the **DesignerItem** class has to implement this interface to become part of the grouping infrastructure, which works like this:

- Create a new **DesignerItem** object with a unique **ID** and with its **IsGroup** property set to **true**
- For each group member, set the **ParentID** to the **ID** of the group parent.

This is simple, but the real work happens when I modify items (Select, Move, Resize, Copy, ...); with each of these operations I have to consider an item's group status. Sounds like a lot of work, but it's not as painful as it would be without LINQ. For this, I have wrapped most of the work into the **SelectionService** class.

Note: The **Connection** class does not implement the **IGroupable** interface and so cannot directly be part of a group, but indirectly - since a connection is always attached to an item. This gives me the flexibility to re/connect items, no matter if they are members of a group or not.

Save

To save a diagram, I have chosen to use a combination of XML and XAML. For the **DesignerItem** related data I use XML, and the content is serialized to XAML. Here again, please note that serializing a designer item's content to XAML only preserves the visual aspects and thus is used as a short term solution only. To create the XML file, I use LINQ. Since this is the first time I experiment with LINQ, don't expect it to be necessarily the "right" way to use it.

Here is an example of how I serialize designer items:

```
XElement serializedItems = new XElement("DesignerItems",
    from item in designerItems
    let contentXaml = XamlWriter.Save(((DesignerItem)item).Content)
    select new XElement("DesignerItem",
        new XElement("Left", Canvas.GetLeft(item)),
        new XElement("Top", Canvas.GetTop(item)),
        new XElement("Width", item.Width),
        new XElement("Height", item.Height),
        new XElement("ID", item.ID),
        new XElement("zIndex", Canvas.GetZIndex(item)),
        new XElement("IsGroup", item.IsGroup),
        new XElement("ParentID", item.ParentID),
        new XElement("Content", contentXaml)
    )
);
```

The **let** keyword allows you to store the result of a sub-expression in a variable that can be used in a subsequent expression. Here I use this feature to save the serialized content in the **contentXaml** variable, which I use a few lines below. Finally, I use the **Save** method of the **XElement** class to store the element's underlying XML tree:

```
XElement.Save(fileName)
```

Open

When loading a diagram from an XML file, we have to start with the designer items because we need their connectors to create connections. We have learned that connectors are part of the item's template, so the designer item has to load its template before we can continue. Fortunately the **Control** class provides the **ApplyTemplate()** method which forces the WPF layout system to load the control template so that its parts can be referenced.

In the previous article, I provided a mechanism to customize the **ConnectorDecorator** template, which allows you to freely position connectors around a designer item. That solution did apply the customized template after the designer item's **Loaded** event was fired and that event is not fired before the item becomes visible on your screen. Now the screen cannot be redrawn before the command has ended. So the only way is to set the customized **ConnectorDecorator** template explicitly within the **Open** command, see the **SetConnectorDecoratorTemplate(item)** method.

Note: When defining customized connectors, you must set the **x:Name** property. A connection uses the name to identify its source and sink connectors.

```
<s:Connector x:Name="Left" Orientation="Left"
    VerticalAlignment="Center" HorizontalAlignment="Left"/>
```

Copy, Paste, Delete, Cut

The **Copy** and **Paste** commands work analogous to the **Open** and **Save** commands, except that they are applied only to the selected items and that they read and write the serialized content to the **Clipboard**. The **Delete** command simply removes all selected items from the designer canvas' **Children** collection, and the **Cut** command finally is a combination of **Copy** and **Delete** command.

Align, Distribute

Not much to say about these commands, except that the reference item for alignment is the item that was selected at first (also called primary selection). This works only when you select items with the LeftMouseButton + Ctrl, or LeftMouseButton + Shift, but not if you use rubberband selection.

Order

The **Panel** class (from which **Canvas** is derived) provides an attached property named **ZIndex** that defines the order on the z-plane in which the children appear, so we only have to change that property to bring an item forward or backward.

History

- 25th March, 2008 -- Original version submitted

License

This article, along with any associated source code and files, is licensed under [The Code Project Open License \(CPOL\)](#)

About the Author




sukram

Austria 

No Biography provided

Comments and Discussions

 **275 messages** have been posted for this article Visit <http://www.codeproject.com/Articles/24681/WPF-Diagram-Designer-Part-4> to post and view comments on this article, or click [here](#) to get a print view with messages.

[Permalink](#) | [Advertise](#) | [Privacy](#) | [Mobile](#)
Web03 | 2.6.130611.1 | Last Updated 26 Mar 2008

Article Copyright 2008 by sukram
Everything else Copyright © [CodeProject](#), 1999-2013
[Terms of Use](#)