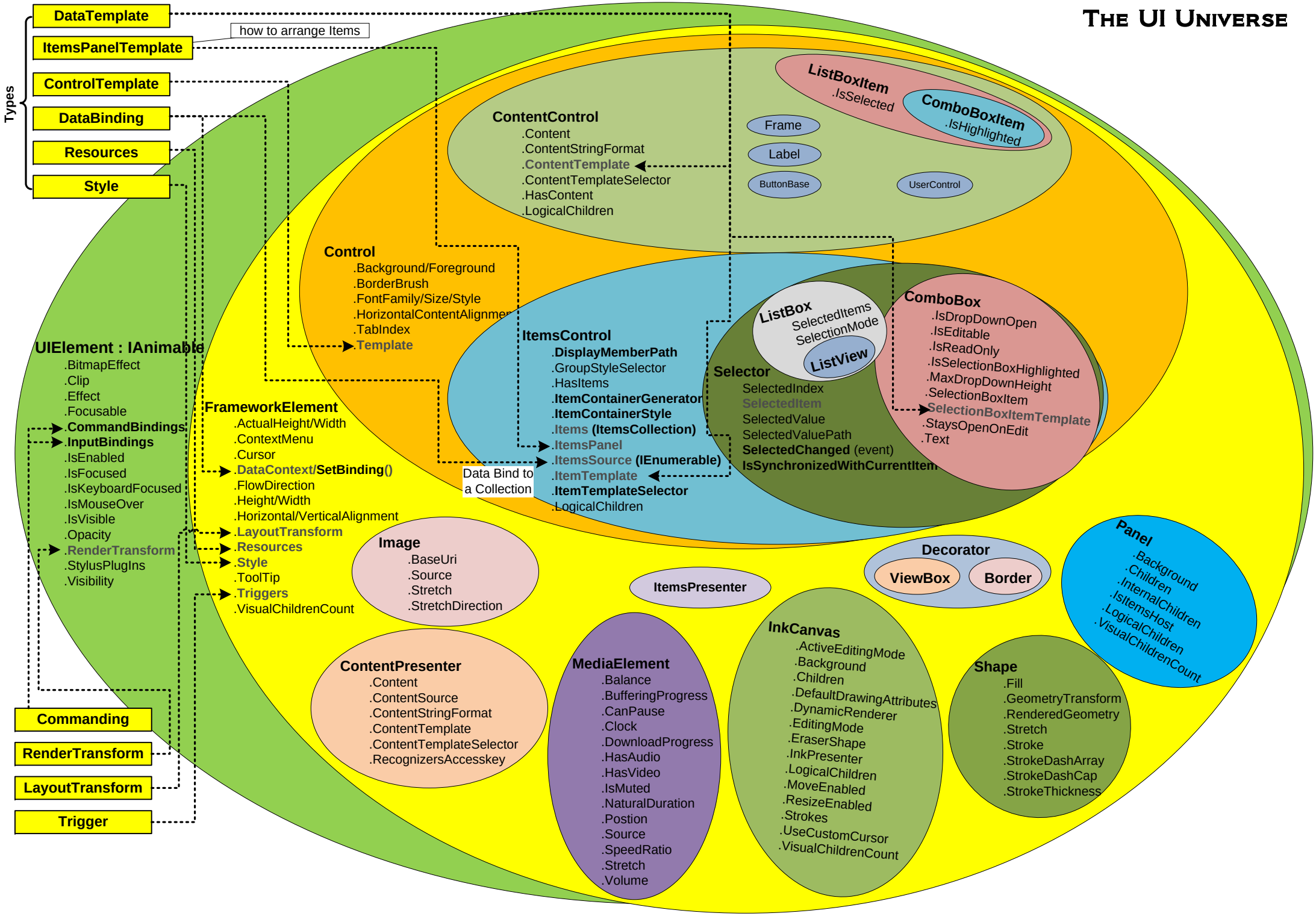
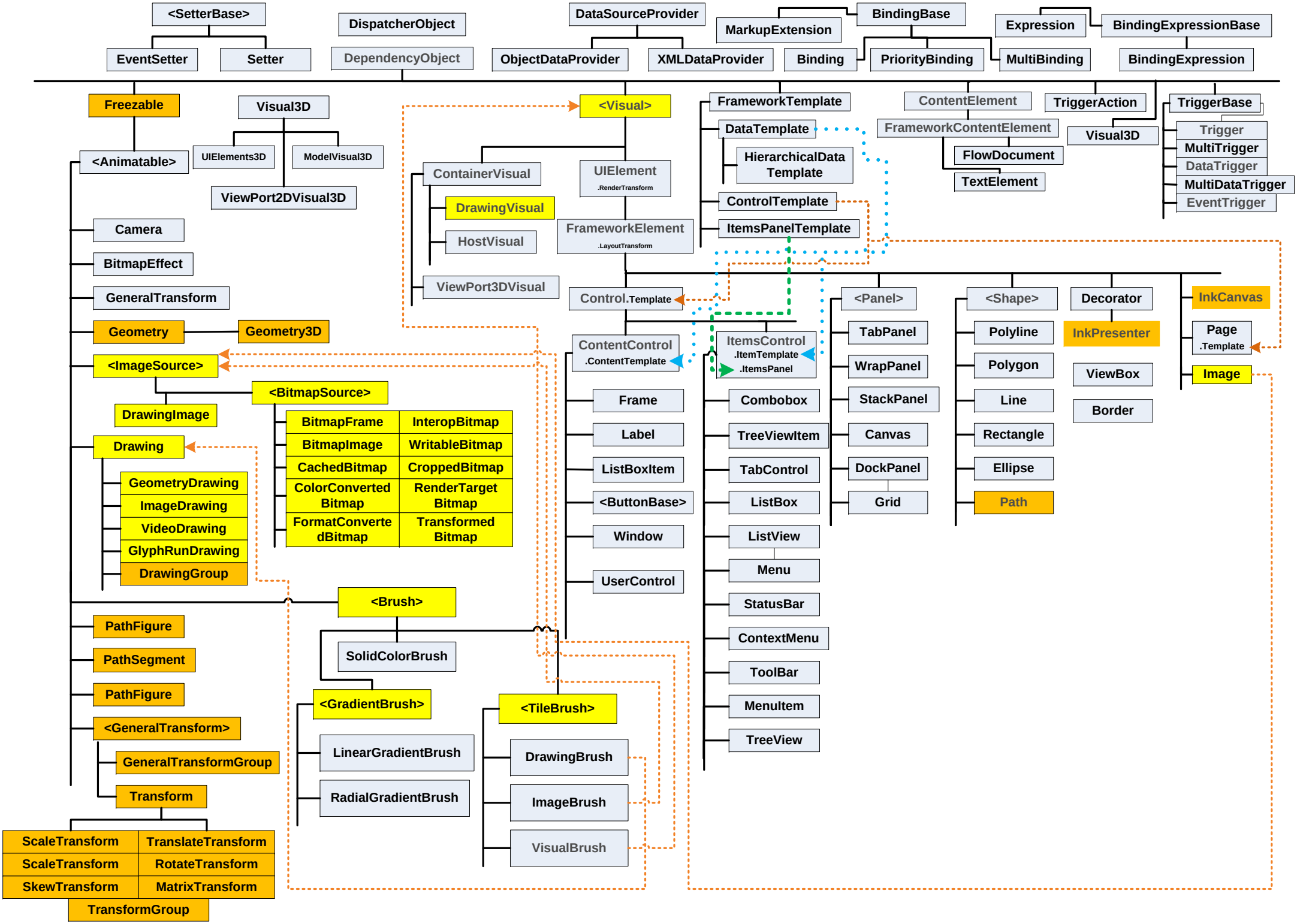
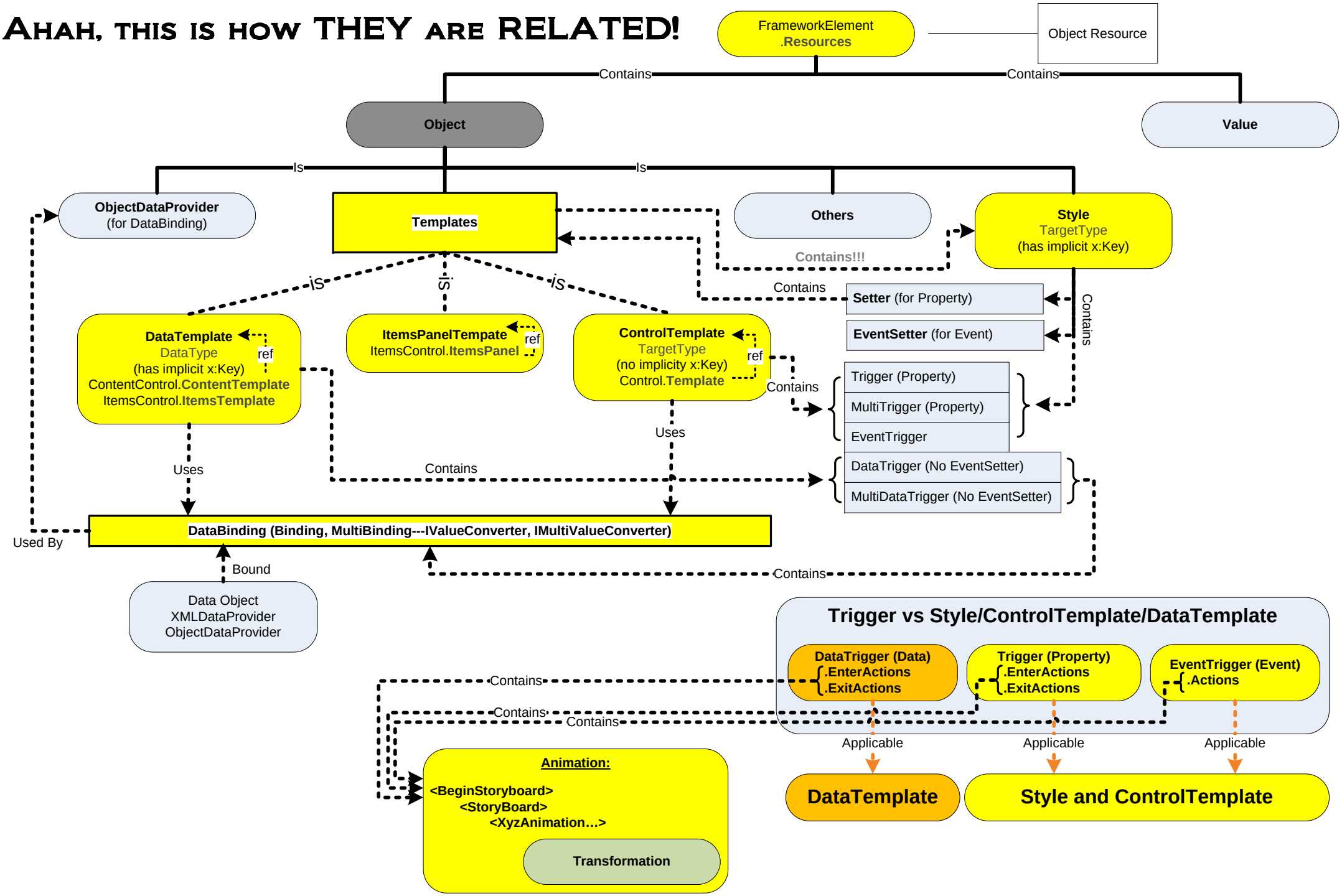
The UI Universe

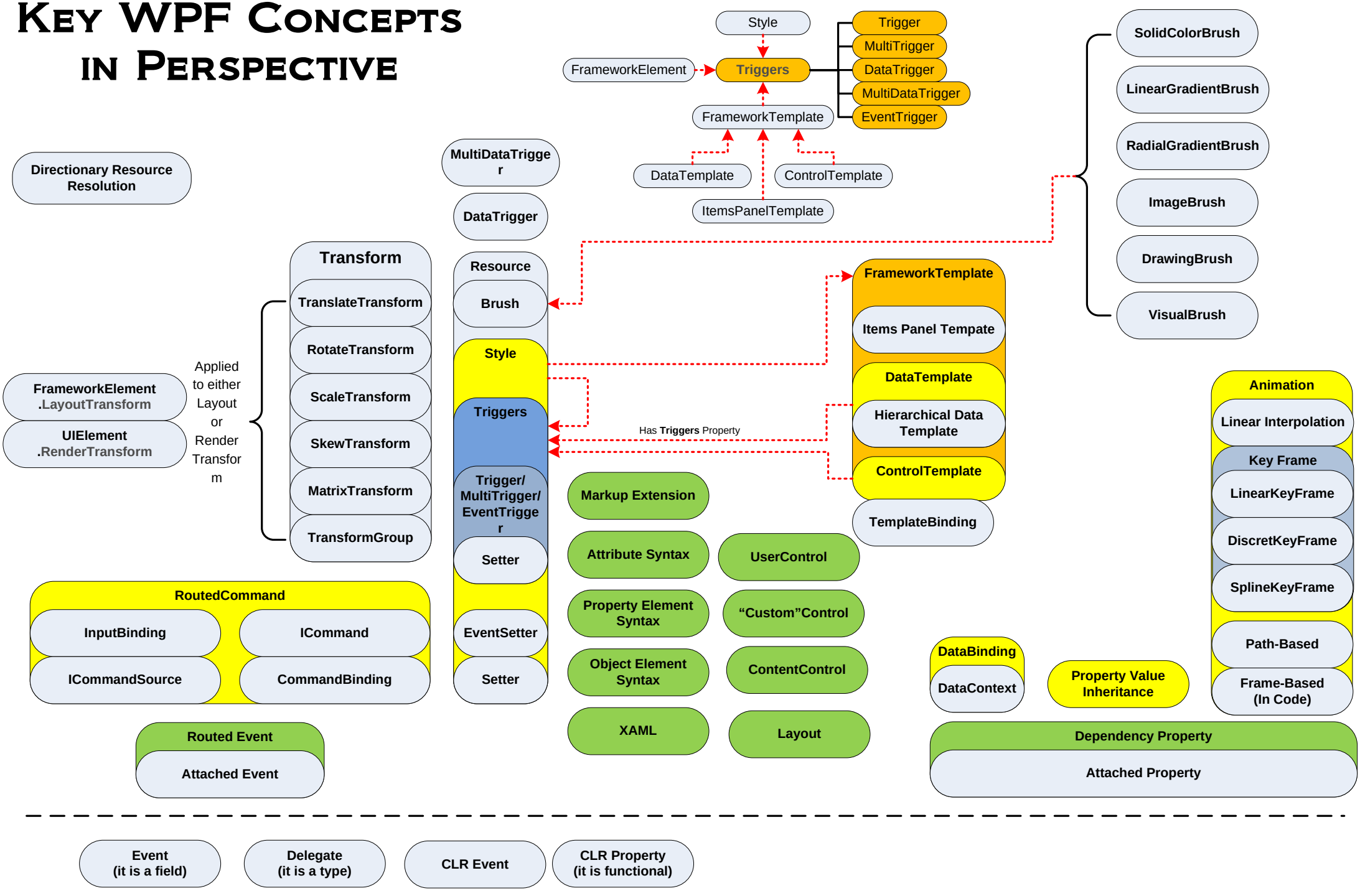



<SetterBase>
EventSetter
Setter
DispatcherObject
DependencyObject
DataSourceProvider
ObjectDataProvider
XMLDataProvider
MarkupExtension
BindingBase
Binding
PriorityBinding
Expression
MultiBinding
BindingExpressionBase
BindingExpression
Freezable
<Animatable>
Visual3D
UIElements3D
ModelVisual3D
ViewPort2DVisual3D
Camera
BitmapEffect
GeneralTransform
Geometry
Geometry3D
<ImageSource>
DrawingImage
Drawing
GeometryDrawing
ImageDrawing
VideoDrawing
GlyphRunDrawing
DrawingGroup
<BitmapSource>
BitmapFrame
InteropBitmap
BitmapImage
WritableBitmap
CachedBitmap
CroppedBitmap
ColorConverted Bitmap
RenderTarget Bitmap
FormatConverte dBitmap
Transformed Bitmap
<Brush>
PathFigure
PathSegment
PathFigure
<GeneralTransform>
GeneralTransformGroup
Transform
ScaleTransform
TranslateTransform
ScaleTransform
RotateTransform
SkewTransform
MatrixTransform
TransformGroup
SolidColorBrush
<GradientBrush>
LinearGradientBrush
RadialGradientBrush
<TileBrush>
DrawingBrush
ImageBrush
VisualBrush
<Visual>
ContainerVisual
DrawingVisual
HostVisual
ViewPort3DVisual
UIElement
.RenderTransform
FrameworkElement
.LayoutTransform
Control.Template
ContentControl
.ContentTemplate
Frame
Label
ListBoxItem
<ButtonBase>
Window
UserControl
ItemsControl
.ItemTemplate
.ItemsPanel
Combobox
TreeViewItem
TabControl
ListBox
ListView
Menu
StatusBar
ContextMenu
ToolBar
MenuItem
TreeView
FrameworkTemplate
DataTemplate
HierarchicalData Template
ControlTemplate
ItemsPanelTemplate
<Panel>
TabPanel
WrapPanel
StackPanel
Canvas
DockPanel
Grid
ContentElement
FrameworkContentElement
FlowDocument
TextElement
<Shape>
Polyline
Polygon
Line
Rectangle
Ellipse
Path
TriggerAction
Visual3D
Decorator
InkPresenter
ViewBox
Border
TriggerBase
Trigger
MultiTrigger
DataTrigger
MultiDataTrigger
EventTrigger
InkCanvas
Page
.Template
Image

Ahah, this is how THEY are RELATED!



Key WPF Concepts in Perspective



Resources (Object)

What are Resources?

A mechanism to reuse predefined objects and values.

```
<Page Name="root"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">

  <Page.Resources>
    <SolidColorBrush x:Key="MyBrush" Color="Gold"/>  // A Brush Resource

    <Style TargetType="Border" x:Key="PageBackground"> // A Style Resource
      <Setter Property="Background" Value="Blue"/>
    </Style>

    <Style TargetType="TextBlock" x:Key="TitleText"> // Another Style Resource
      <Setter Property="Background" Value="Blue"/>
      <Setter Property="DockPanel.Dock" Value="Top"/>
      <Setter Property="FontSize" Value="18"/>
      <Setter Property="Foreground" Value="#4E87D4"/>
      <Setter Property="FontFamily" Value="Trebuchet MS"/>
      <Setter Property="Margin" Value="0,40,10,10"/>
    </Style>

    <Style TargetType="TextBlock" x:Key="Label">  // Yet, another Style Resource
      <Setter Property="DockPanel.Dock" Value="Right"/>
      <Setter Property="FontSize" Value="8"/>
      <Setter Property="Foreground" Value="{StaticResource MyBrush}"/>
      <Setter Property="FontFamily" Value="Arial"/>
      <Setter Property="FontWeight" Value="Bold"/>
      <Setter Property="Margin" Value="0,3,10,0"/>
    </Style>

  </Page.Resources>

  <StackPanel>
    <Border Style="{StaticResource PageBackground}">
      <DockPanel>
        <TextBlock Style="{StaticResource TitleText}">Title</TextBlock>
        <TextBlock Style="{StaticResource Label}">Label</TextBlock>

        <TextBlock
           DockPanel.Dock="Top" HorizontalAlignment="Left" FontSize="36"
           Foreground="{StaticResource MyBrush}" Text="Text" Margin="20" />

        <Button
           DockPanel.Dock="Top" HorizontalAlignment="Left" Height="30"
           Background="{StaticResource MyBrush}" Margin="40">Button</Button>

        <Ellipse
           DockPanel.Dock="Top" HorizontalAlignment="Left" Width="100"
           Height="100" Fill="{StaticResource MyBrush}" Margin="40" />

      </DockPanel>
    </Border>
  </StackPanel>
</Page>
```

Resources Dictionary

StaticResource vs DynamicResource:

When you use a markup extension, you typically provide one or more parameters in string form that are processed by that particular markup extension, rather than being evaluated in the context of the property being set. The StaticResource Markup Extension processes a key by looking up the value for that key in all available resource dictionaries. This happens during loading, which is the point in time when the loading process needs to assign the property value that takes the static resource reference. The DynamicResource Markup Extension instead processes a key by creating an expression, and that expression remains unevaluated until the application is actually run, at which time the expression is evaluated and provides a value.

Resources in Code:

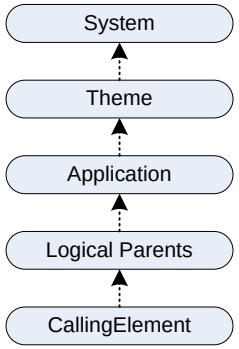
Accessing:

```
FrameworkElement.FindResource("<resKey>");
FrameworkElement.TryFindResource("<resKey>");
SetResourceReference("<resKey>", DependencyProperty);
resDict["resKey"]; // no normal Resource Resolution is performed!
```

Creation:

```
var resDict = new ResourceDictionary();
FrameworkElement.Resources = resDict;
resDict.Add(…);
```

Resource Resolution:



Merged Resource Dictionaries:

FrameworkElements have a Resources Property which references to the Main Resource Dictionary, which contains a MergedDictionaries, whose lookup scope is after that of the Main Resource Dictionary.

```
<Page.Resources>
  <ResourceDictionary>

    <ResourceDictionary.MergedDictionaries>
      <ResourceDictionary Source="myresourcedictionary.xaml"/>
      <ResourceDictionary Source="myresourcedictionary2.xaml"/>
    </ResourceDictionary.MergedDictionaries>

    <!-- Other Resources can be defined here after -->

  </ResourceDictionary>
</Page.Resources>
```

There are couple of Resource in WPF:

= Assembly Resources
= System Resources (SystemColors, SystemFonts, SystemParameters)

eg: {x:Static SystemColors.WindowTextColor}

= Application Resources

```
<Label Background="{DynamicResource {x:Static SystemColors.xxxKey}}"/>
```

Style (Like CSS, but much more Powerful)

Named Style:

```
<Page.Resources>

  <Style x:Key="WinterStyle">
    <Setter Property="Control.Background" Value="BlueViolet" />
  </Style>

  <Style x:Key="SummerStyle">
    <Setter Property="Control.Background" Value="Yellow" />
  </Style>

</Page.Resources>
```

Named Style Use:

```
<Button x:Name="GoButtonWithStyle" Style="{StaticResource WinterStyle}">
  "Go"
</Button>
```

Chosing Style Programmatically:

```
<Button x:Name="WelcomeButton"  Content="Welcome user!" ></Button>
```

In the code, an event that fires when the page loads, is added:

```
void Page_Load(object sender, RoutedEventArgs e) {
   if (DateTime.Now.Month > 8 || DateTime.Now.Month < 3)
      WelcomeButton.Style = (Style)FindResource("WinterStyle");
   else
      WelcomeButton.Style = (Style)FindResource("SummerStyle");
}
```

Targetd Style: (Automatically Applied to Types specified in **TargetType** Attribute)

```
<Page.Resources>

  <Style TargetType="{x:Type Button}">
    <Setter Property="Background" Value="BlueViolet" />
    <Setter Property="FontSize" Value="20" />
    <Setter Property="Width" Value="50" />
  </Style>

  <Style TargetType="{x:Type TextBox}">
    <Setter Property="Background" Value="White" />
    <Setter Property="Width" Value="200" />
    <Setter Property="Height" Value="30" />
  </Style>
</Page.Resources>

<StackPanel>
  <Button x:Name="GoButton1" Content="Go" ></Button>
  <TextBox x:Name="EmptyTextBox" Text="Hello!"></TextBox>
</StackPanel>
```

Automatically Applied Based On Type

A Style is used to affect the elements on which it is applied, is normally defined in FrameworkElement.Resources, which most likely are "Windows.Resources and Application.Resources."

Can be Named, by "x:Key", or Targeted, by "TargetType" Property.

Targeted Styles are applied automatically for elements of the same or sub-type. Because "TargetType" establish a context, thus in specifying Properties, ClassName is not needed.

Named Styles must use ClassName in specifying Properties.

Setters is the Default Property of Style Class, thus it can be omitted in most cases, allowing specifying <Setter> directly right under </Setter>

Setters has a Collection of <Setter>'s; Each <Setter> has Property and Value Attributes.

EventSettter has Event and Handler Attributes

Triggers collection provides conditional Style

Simple <Trigger> has Property and Value Attributes, followed by collection of <Trigger.Setters>
MultiTrigger has a Conditions Property, a Collection of <Condition> which has Property and Value Attributes

EventTrigger has RoutedEvent Attribute and Actions Property, a Collection

BasedOn Attribute can be used to inherit a Style

Trigger (Trigger, DataTrigger, EventTrigger) also has EnterActions and ExitActions Attribute inherited from TriggerBase class.

Style Inheritance:

```
<Page.Resources>

  <Style x:Key="ButtonBaseStyle">
    <Setter Property="Button.Background" Value="BlueViolet" />
    <Setter Property="Button.Height" Value="30" />
  </Style>

  <Style x:Key="ButtonDerivedStyle" BasedOn="{StaticResource ButtonBaseStyle}">
    <Setter Property="Button.Foreground" Value="FloralWhite" />
  </Style>

</Page.Resources>

<StackPanel>
  <Button x:Name="GoButtonWithBaseStyle"
    Style="{StaticResource ButtonBaseStyle}"  Content="Go" >
  </Button>
  <Button x:Name="GoButtonWithDerivedStyle"
    Style="{StaticResource ButtonDerivedStyle}"  Content="Go" >
  </Button>
</StackPanel>
```

Styling Example

Examples: ObjectDataProvider, Targeted Style, Named Style, Style Inheritance, ControlTemplate in Targeted Style (so it can be automatically applied), DataTemplate, Triggers in Style, DataBinding both in Element and in DataTemplate.

```
<Window x:Class="StylingIntroSample.Window1"
   xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
   xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
   xmlns:local="clr-namespace:StylingIntroSample"
   Loaded="WindowLoaded"
   Title="StylingIntroSample" SizeToContent="WidthAndHeight">

 <Window.Resources>
  <ObjectDataProvider
   x:Name="PhotosODP" x:Key="MyPhotos"
   ObjectType="{x:Type local:PhotoList}"/>

  <!--A Style that affects all TextBlocks-->
  <Style TargetType="TextBlock">
   <Setter Property="HorizontalAlignment" Value="Center" />
   <Setter Property="FontFamily" Value="Comic Sans MS"/>
   <Setter Property="FontSize" Value="14"/>
  </Style>

  <!--A Style that extends the previous TextBlock Style-->
  <!--This is a "named style" with an x:Key of TitleText-->
  <Style BasedOn="{StaticResource {x:Type TextBlock}}"
       TargetType="TextBlock" x:Key="TitleText">

   <Setter Property="FontSize" Value="26"/>
   <Setter Property="Foreground">
   <Setter.Value>
     <LinearGradientBrush StartPoint="0.5,0" EndPoint="0.5,1">
      <LinearGradientBrush.GradientStops>
       <GradientStop Offset="0.0" Color="#90DDDD" />
       <GradientStop Offset="1.0" Color="#5BFFFF" />
      </LinearGradientBrush.GradientStops>
     </LinearGradientBrush>
    </Setter.Value>
   </Setter>
  </Style>

  <!--Horizontal ListBox Control Template-->
  <Style TargetType="ListBox">
   <Setter Property="Template">
    <Setter.Value>
     <ControlTemplate TargetType="ListBox">
      <Border CornerRadius="5"
           Background="{TemplateBinding ListBox.Background}">
       <ScrollViewer HorizontalScrollBarVisibility="Auto">
        <StackPanel Orientation="Horizontal"
                 VerticalAlignment="Center"
                 HorizontalAlignment="Center"
                 IsItemsHost="True"/>
       </ScrollViewer>
      </Border>
     </ControlTemplate>
    </Setter.Value>
   </Setter>
  </Style>

  <!--DataTemplate to display Photos as images, not as Path strings-->
  <DataTemplate DataType="{x:Type local:Photo}">
   <Border Margin="3">
    <Image Source="{Binding Path=Source}"/>
   </Border>
  </DataTemplate>

  <Style TargetType="ListBoxItem">
   <Setter Property="Opacity" Value="0.5" />
   <Setter Property="MaxHeight" Value="75" />
   <Style.Triggers>
    <Trigger Property="IsSelected" Value="True">
     <Trigger.Setters>
      <Setter Property="Opacity" Value="1.0" />
     </Trigger.Setters>
    </Trigger>

    <EventTrigger RoutedEvent="Mouse.MouseEnter">
     <EventTrigger.Actions>
      <BeginStoryboard>
       <Storyboard>
        <DoubleAnimation
         Duration="0:0:0.2"
         Storyboard.TargetProperty="MaxHeight"
         To="90"  />
       </Storyboard>
      </BeginStoryboard>
     </EventTrigger.Actions>
    </EventTrigger>

    <EventTrigger RoutedEvent="Mouse.MouseLeave">
     <EventTrigger.Actions>
      <BeginStoryboard>
       <Storyboard>
        <DoubleAnimation
         Duration="0:0:1"
         Storyboard.TargetProperty="MaxHeight"  />
       </Storyboard>
      </BeginStoryboard>
     </EventTrigger.Actions>
    </EventTrigger>

   </Style.Triggers>
  </Style>

 </Window.Resources>

 <StackPanel Margin="10">

  <TextBlock Style="{StaticResource TitleText}" Name="textblock1">
     My Pictures
  </TextBlock>

  <TextBlock>Check out my new pictures!</TextBlock>

  <ListBox ItemsSource="{Binding Source={StaticResource MyPhotos}}"
         Background="Silver" Width="600" Margin="10" SelectedIndex="0"/>
 </StackPanel>

</Window>
```

Default DataTemplate For Photo Object

Implicit Binding Source Is Specified By Datatype

Default Style for ListBoxItem

Default Style For TextBlock, Targeted Style

Inherited From

Default Style For ListBox

Control Template For ListBox

Local Style Specification, Named Style

Data Bi… The Wi… Data So…

<Style TargetType="TextBlock"> vs <Style TargetType="{x:Type TextBlock}">

TargetType vs DataType

TargetType used in ControlTemplate

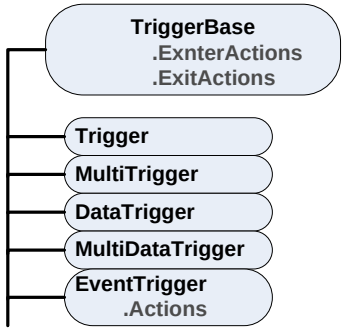
DataType used in DataTemplate

Triggers (Live inside Styles and Templates)

Property Trigger (Trigger/MultiTrigger)

```
<Style TargetType="{x:Type Button}">

  <Setter Property="Button.Background" Value="AliceBlue" />
  <Setter Property="Button.Opacity" Value="0.5" />

  <Style.Triggers>

    <Trigger Property="IsMouseOver" Value="True">
      <Setter Property="Button.Opacity" Value="1"/>
      <Setter Property="Button.Background" Value="Green"/>
    </Trigger>

    <Trigger Property="IsEnabled" Value="False">
      <Setter Property="Button.Background" Value="Yellow"/>
    </Trigger>

  </Style.Triggers>
</Style>
```

```
<Style TargetType="{x:Type Button}">

  <Setter Property="Button.Background" Value="AliceBlue" />

  <Style.Triggers>

    <MultiTrigger>

      <MultiTrigger.Conditions>
        <Condition Property="IsMouseOver" Value="True"/>
        <Condition Property="IsEnabled" Value="True" />
      </MultiTrigger.Conditions>

      <Setter Property="Button.Background" Value="Yellow"/>

    </MultiTrigger>

  </Style.Triggers>

</Style>
```



DataTrigger

```
namespace Demo{
  public class User { // defining the User Class
    public User(string name, string role) {
      this._name = name;
      this._role = role;
    }
    public string Name {get;set}
    public string Role {get;set}
  }
}

public class Users : ObservableCollection<User> {
  public Users() { // the Default Constructor
    this.Add(new User("Gill Cleeren", "Admin"));
    this.Add(new User("Steve Smith", "Contributor"));
    this.Add(new User("John Miller", "User"));
  }
}

<Page x:Class="Demo.Page1"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="Page1"
  xmlns:clr="clr-namespace:Demo>

  <Page.Resources>

    <clr:Users x:Key="myUsers" /> // instantiate the Users

    <DataTemplate DataType="{x:Type clr:User}">
      <TextBlock Text="{Binding Path=Name}"/>
    </DataTemplate>                    Binding Source is

    <Style TargetType="{x:Type ListBoxItem}">
                                  Binding Property of DataTrigger
      <Style.Triggers>          Binding Markup Extension, Binding Source is
        <DataTrigger Binding="{Binding Path=Role}" Value="Admin">
          <Setter Property="Foreground" Value="Red" />
        </DataTrigger>
      </Style.Triggers>

    </Style>

  </Page.Resources>

  <StackPanel>
    <ListBox                                Data Bind to User Collection
      Width="200"
      ItemsSource="{Binding Source={StaticResource myUsers}}" />
  </StackPanel>
</Page>
```

EventTrigger

EnterActions
ExitActions

```
<Window.Resources>

  <Style TargetType="{x:Type Button}">

    <Setter Property="Width" Value="200" />
    <Setter Property="Height" Value="100" />
    <Setter Property="Margin" Value="20" />
    <Setter Property="HorizontalAlignment" Value="Left" />

    <Style.Triggers>

      <EventTrigger RoutedEvent="Button.MouseEnter">

        <EventTrigger.Actions>
          <BeginStoryboard>
            <Storyboard>
              <DoubleAnimation
                To="300" Duration="0:0:3"
                Storyboard.TargetProperty="(Button.Width)" />

                        <DoubleAnimation To="200" Duration="0:0:3"
                Storyboard.TargetProperty="(Button.Height)" />

            </Storyboard>
          </BeginStoryboard>
        </EventTrigger.Actions>
      </EventTrigger>

      <EventTrigger RoutedEvent="Button.MouseLeave">
        <EventTrigger.Actions>
          <BeginStoryboard>
            <Storyboard>
              <DoubleAnimation
                Duration="0:0:3"
                Storyboard.TargetProperty="(Button.Width)" />

                        <DoubleAnimation  Duration="0:0:3"
                Storyboard.TargetProperty="(Button.Height)" />

            </Storyboard>
          </BeginStoryboard>
        </EventTrigger.Actions>
      </EventTrigger>
    </Style.Triggers>
  </Style>

</Window.Resources>

<StackPanel>
  <Button x:Name="GrowButton" Content="Hello MSDN" />
</StackPanel>
```

DataBinding

System.Windows.Data

Basics:

1) What is it? It allows a DependencyObject (**target**) be bound to a piece of data (**source**)
2) Features: (From a FrameworkElement standpoint)
   - A) Binding to Collection: A ListBox is bound to an object collection. (Use "/" in PATH to specify current item)
   - B) Binding to an Element
   - C) Binding Non-Visual Object, XML Data, Collections of objects.
   - B) CollectionView/Source: It can be used to group, sort, and filter the underlining Collection, without modifying it; thus one Collection can have multiple CollectionViews.
   - C) Master-Detail
   - D) Data Conversion, through Converters implement IValueConverter or IMultiValueConverter interface.
   - E) Data Validation (ValidationRule)
   - F) Function (via DataSourceProvider)
   - G) DataBase Table, WebServices, the Whole Object (without specifying the Path)
3) **ObjectDataProvider** and **XMLDataProvider** can be used to make Any Custom Object usable as Binding Source.
4) Debugging: Use the attached property PresentationTraceSources.TraceLevel="High" on a binding object.

```
<Binding Source="{StaticResource myDataSrc}" Path="ColorName"
      diag:PresentationTraceSources.TraceLevel="High"/>
```

5) **MultiBinding** is supported.

DataContext Chaining using Converter:

On WPF logic tree, there can be more than one DataContexts, which can be chained, producing a DataContext from its ancestor using Converter. This is useful in consolidating multiple conversion for the same Data object, allowing direct binding to new DataContext's properties directly.

http://jobijoy.blogspot.com/2008/09/xaml-clock.html

```
<Window DataContext="{x:Null}" …> // set in code behind
    …
    <Canvas
        DataContext=
        "{Binding Converter={StaticResource myConverter}}" >
        ...
```

Creating Binding:

In XAML: (using DataContext)

```
<DockPanel>
    <DockPanel.Resources>
        <c:MyData x:Key="myDataSrc"/>
    </DockPanel.Resources>
    <DockPanel.DataContext>
        <Binding Source="{StaticResource myDataSrc}"/>
    </DockPanel.DataContext>

    <Button Background="{Binding Path=ColorName}">
        Set My Background
    </Button>
</DockPanel>
```

In XAML: (using Binding, Markup Extension)

```
<DockPanel>
    <DockPanel.Resources>
        <c:MyData x:Key="myDataSrc"/>
    </DockPanel.Resources>
    <Button Background=
            "{Binding Source={StaticResource myDataSrc},
            Path=ColorName}">
        Set My Background
    </Button>
</DockPanel>
```

In Code (C#):

```
var myData = new MyData();
var binding = new Binding();
binding.Source = myData;
binding.Path = "ColorName";
var btn = new Button();
btn.SetBinding("BackgroundProperty", binding);
```

<Binding> vs {Binding…}
<Binding> is Binding object
{Binding…} Markup Extension

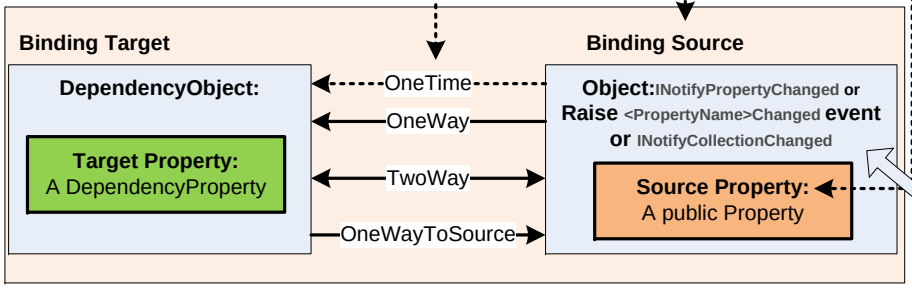
Binding Object:

Mutual Exclusive:
- **.Source**, to referece CLR object (including object Collection), ADO.Net data, XML data, DependencyObject
- **.ElementName**, to specify an Element
- **.RelativeSource**, to be used in a Template or Style

- **.Path**, to specify the Source Property (not needed if Binding to the entire object, particularly object Collection or Binding to a string object, or wanting to use the Converter to convert an object's Multi-Property into desired object)
- **.XPath**, to specify XML data
- **.Mode**, if not specified, Default Mode is used.
- **.UpdateSourceTrigger**, when to update Source in TwoWay and OneWayToSource.
- **.ValidationRules**, to validate Target Property Data
- **.Converter**, to convert data to target format
- **.BindingDirectlyToSource**, when "true", bind to the DataSourceProvider (eg: ObjectDataProvider).

Note: Target Property has a Default Mode Value

DataBinding Diagram: Concepts



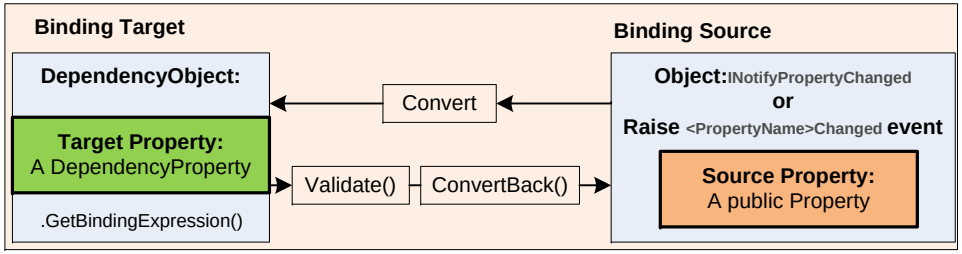
Wow: Raising PropertyChanged event without specifying PropertyName means to tell WPF to check all Properties!

BindingOperations.GetBindingExpressoin()

FrameworkElement.GetBindingExpression()

BindingExpression
- .UpdateSource()
- .UpdateTarget()
- .DataItem

DataBinding Diagram: Converter and Validation



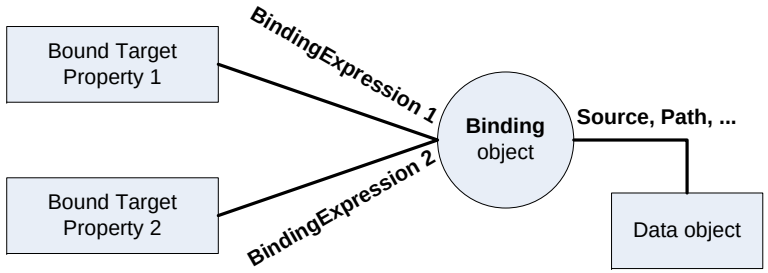
What Triggers Source Updates

public enum UpdateSourceTrigger:

- **Default**, Target Property's default.
- **PropertyChanged**
- **LostFocus**
- **Explicit**, Target Property need to explicitly call UpdateSource() on BindingExpression to update Source.

Binding vs BindingExpression:

Binding class contains all Binding Source info that can be shared across several BindingExpressions which can not be shared because they are unique to their own instance of Binding.



CustomValidation: ValidationRule

public override ValidationResult **Validate**(object value, …)

CustomConverter: IValueConverter

public object **Convert**(object value, Type targetType, …)
public object **ConvertBack**(object value, Type targetType, …)

http://www.nbdtech.com/Free/WpfBinding.pdf

http://www.interact-sw.co.uk/iangblog/2007/09/03/bindtocurrent

Data Binding in InkCanvas

ms-help:HowTo: Data Bind to an InkCanvas

```
<Canvas>
  <Canvas.Resources>
    <!--Define an array containing the InkEditingMode Values.-->
    <x:Array x:Key="MyEditingModes" Type="{x:Type InkCanvasEditingMode}">
      <x:Static Member="InkCanvasEditingMode.Ink"/>
      <x:Static Member="InkCanvasEditingMode.Select"/>
      <x:Static Member="InkCanvasEditingMode.EraseByPoint"/>
      <x:Static Member="InkCanvasEditingMode.EraseByStroke"/>
    </x:Array>

    <!--Define an array containing some DrawingAttributes.-->
    <x:Array x:Key="MyDrawingAttributes" Type="{x:Type DrawingAttributes}">
      <DrawingAttributes Color="Black" FitToCurve="true" Width="3" Height="3"/>
      <DrawingAttributes Color="Blue" FitToCurve="false" Width="5" Height="5"/>
      <DrawingAttributes Color="Red" FitToCurve="true" Width="7" Height="7"/>
    </x:Array>

    <!--Create a DataTemplate to display the DrawingAttributes shown above-->
    <DataTemplate DataType="{x:Type DrawingAttributes}" >
      <Border Width="80" Height="{Binding Path=Height}">
        <Border.Background >
          <SolidColorBrush Color="{Binding Path=Color}"/>
        </Border.Background>
      </Border>
    </DataTemplate>
  </Canvas.Resources>

  <!--
     Bind the first InkCavas' DefaultDrawingAtributes to a Listbox, called
     lbDrawingAttributes, and its EditingMode to a ListBox called
     lbEditingMode.
  -->
  <InkCanvas Name="ic" Background="LightGray" Canvas.Top="0" Canvas.Left="0" Height="400" Width="200"
          DefaultDrawingAttributes="{Binding ElementName=lbDrawingAttributes, Path=SelectedItem}"
          EditingMode="{Binding ElementName=lbEditingMode, Path=SelectedItem}">
  </InkCanvas>

  <!--
     Bind the Strokes, DefaultDrawingAtributes, and, EditingMode
     properties of the second InkCavas to the first InkCanvas.
  -->

  <InkCanvas Background="LightBlue"  Canvas.Top="0" Canvas.Left="200"  Height="400" Width="200"
          Strokes="{Binding ElementName=ic, Path=Strokes}"
          DefaultDrawingAttributes="{Binding ElementName=ic, Path=DefaultDrawingAttributes}"
          EditingMode="{Binding ElementName=ic, Path=EditingMode}">

    <InkCanvas.LayoutTransform>
      <ScaleTransform ScaleX="-1" ScaleY="1" />
    </InkCanvas.LayoutTransform>

  </InkCanvas>

  <!--Use the array, MyEditingModes, to populate a ListBox-->
  <ListBox Name="lbEditingMode" Canvas.Top="0" Canvas.Left="450" Height="100" Width="100"
          ItemsSource="{StaticResource MyEditingModes}" />

  <!--Use the array, MyDrawingAttributes, to populate a ListBox-->
  <ListBox Name="lbDrawingAttributes" Canvas.Top="150" Canvas.Left="450" Height="100" Width="100"
          ItemsSource="{StaticResource MyDrawingAttributes}" />

</Canvas>
```

Describe How Each DrawingAttributes in the List of DAs is Displayed.

Bound To ListBox of DAs

Bound To ListBox of EditingModes

Bound To A List of EditingModes

Bound to AList of DrawingAttributes

Ideas Behind the WPF Commanding System:

1) **Old way:** Event Handler-based. EventHandler, attached to low-level events (like MouseDown), has all the application logic inside; thus, tightly couple between application logic and events.

2) **Basic ICommand Commanding:** ICommand-based. Instead of use EventHandler, adopt the more abstract concept of Command, which contains application logic in its execute() method. The sequence is: Event triggers Command Execution. In this basic model, we need ICommandSource (CommandSource object) and ICommand (Command object) that implements the command logic. However, because the Command object contains the command logic, command logic is tightly coupled with Command---which could be decoupled with RoutedCommand.

Benefits:

[Do away with Event---No Event is Raised]
[Enable disable state]
[Command object reusable]
[Able to do KeyGesture and MouseGesture to execute Command]

WPF Commanding (Basic Ideas)

(Best) Essential Windows Presentation Foundation – Chris Anderson (Ch 7 Actions)
(Picture speaks volume) Apress - Illustrated WPF by Daniel Solis (Ch 9 Routed Events and Commands)
(Examples Rich) Apress - Pro WPF in C# 2008 by Mathew MacDonald (Ch 10 Commands)

3) **RoutedCommand:** RoutedEvent-based. RoutedCommand does implement ICommand like before; however, it does not contain any command logics, instead its Execute() and CanExecute() methods only are used to raise Routed Events---ExecutedRoutedEvent, PreviewExecutedRoutedEvent; CanExecutedRoutedEvent, PrviewCanExecutedRoutedEvent, respectively. These Routed Events are routed through the Element Tree to look for a CommandBinding object which referring to the matching Command object being executed. It's inside the CommandBinding object that Routed Event Handlers are attached, and it is there that Command Logics are implemented! Thus, separating the Command Logic from the Command object---they are loosely connected through Routed Events.

So far, all are the fundamentals. Now, WPF makes things easier in RoutedCommand (Command), CommandBinding, ICommandSource (CommandSource) areas:

4) **Pre-Built RoutedCommands:** WPF already comes with pre-built RoutedCommands, like ApplicationCommands.Cut; inside these Commands, they already have common gesture configured, so that gestures like Ctrl-X already understood as CommandSource able to trigger the command if it's in enable state. Programmers can use these pre-built Commands directly without the need to create new ones, in most cases.

5) **Built-In CommandBinding** support: Some WPF controls (like TextBox) already have CommandBinding built-in! That means for ApplicationCommands.Cut, a given TextBox already setup to handle it! Programmer just need to designate a CommandSource for the ApplicationCommands.Cut command!

On the Command Source side, WPF also make life easier by:

6) **CommandSource-Ready Controls:** Some WPF controls are already CommandSource because they implement ICommandSource or have InputBindings property. They are:

a) ButtonBase (Button, GridViewColumnHeader, RepeatButton, ToggleButton, CheckBox, RadioButton) [Click]
b) Hyperlink [Click]
c) MenuItem [Click]
d) InputBinding (MouseBinding, KeyBinding) [n/a]
e) ListBoxItem (Through InputBindings) [DoubleClick]

WPF Principles:
1) Element Composition
2) Loose Coupling
3) Declarative Programming

That means Programmers can just associate a Command to them to have them invoke the command when Click or DoubleClick, or in some case, don't need to do anything if a particular command already has built-in support!

Note: **CommandTarget** is ONLY relevant to RoutedCommand; otherwise is ignored. If specified in CommandSource, CommandTarget is where Routed Events are raised; if not specified in CommandSource, CommandTarget is the UIElement with KeyBoard Focus, where Routed Events are raised.

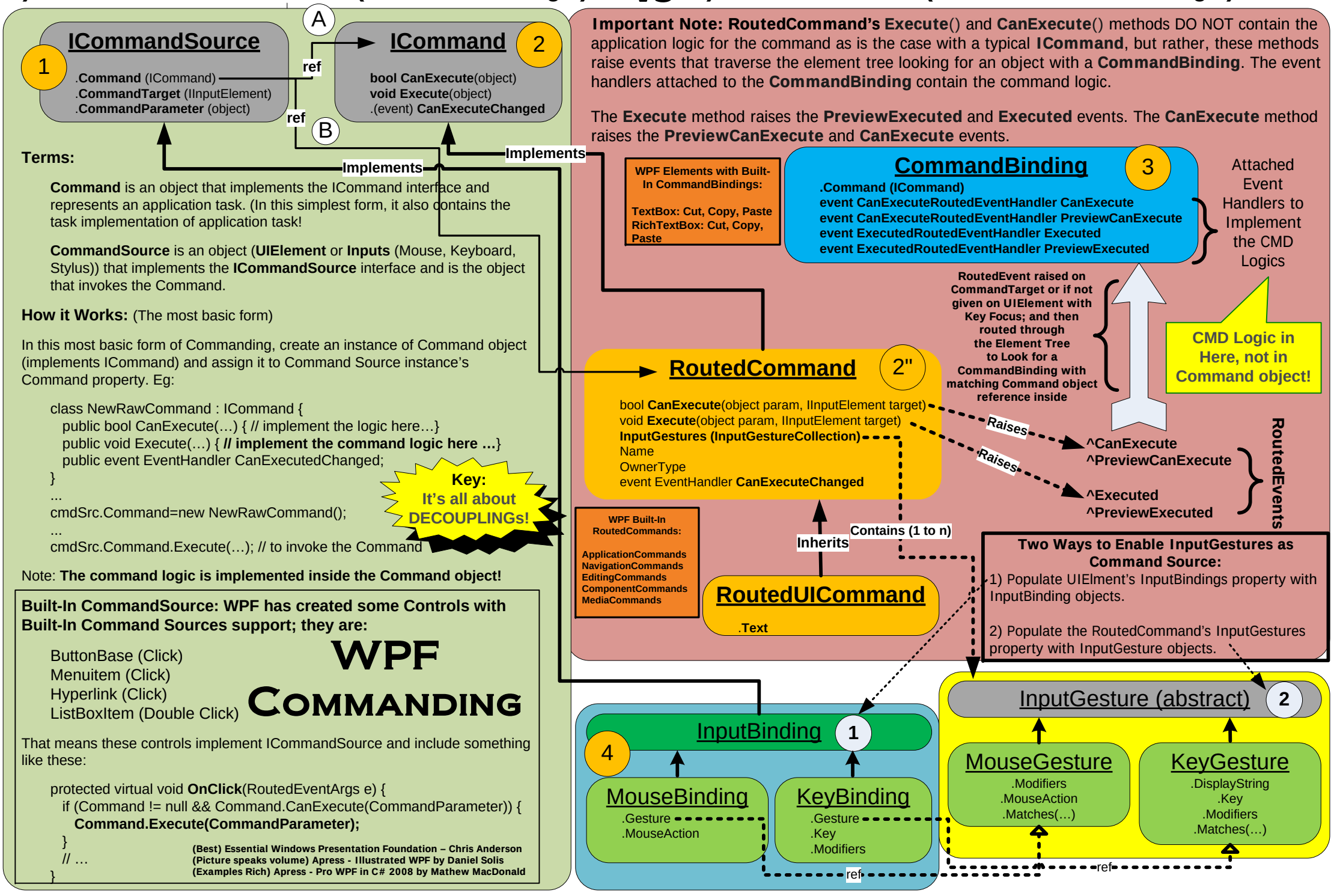
Using Command object to decouple Event from Application Logic (in Command Object)

1) Basic ICommand Command (contains CMD logic)

VS

Using Routed Event to decouple Command object from Application Logic

2) RoutedCommand (doesn't contain CMD logic)



1. A Command can be invoked by two different classes of Command Source:
1a: UI Elements with built-in Command Source support (using Element events)
1b: InputBindings---KeyBinding or MouseBinding (using Keyboard and Mouse)

2. Command source invokes a Command by executing its aCommand.Execute() method, which raises two WPF Routed Events --- Executed and PreviewExecuted.

3. The Executed and PreviewExecuted Routed Events bubble up and tunnel down the Element Tree to locate an UI Element with CommandBinding with corresponding Command in it. If found, Event Handlers are called to execute the Command Logic.

Note:
4. Listening for CommandManager's RequerySuggested (CommandManager.InvalidateREquerySuggested() can force one), a Command raises the CanExecuteChanged event to inform Command Sources that CanExecute state has changed, and it is up to the Command Sources to find out the state by calling a Command.CanExecute() and then decide what to do, enabling or disabling things.

<CommandBinding Command="{x:Static prefix:NewCommands.NewCommand}"
CanExecute="NewCommandCanExecuteHandler"
Executed="NewCommandExecutedHandler"/>

A
B Implements A
B
A
B Derived from A
B

3

CommandBinding:

Elements with Built-In CommandBinding:
TextBox: Cut, Copy, Paste
RichTextBox: Cut, Copy, Paste

Command
^CanExecute
^Executed
^PreviewCanExecute
^PreviewExecuted

Handlers (Command Logics)

Routed through Element Tree

To: Create a NewCommand:
public static var NewCommand = new RoutedCommand();

1

(1 to 1) References

Command:
ICommand

2

CanExecute(o)
Execute(o)
Raises Events On Command Target
^CanExecuteChanged
Raises Events
^CanExecute
^PreviewCanExecute
^Executed
^PreviewExecuted

4

CommandSource:
ICommandSource

1a

Built-In Command Sources:
ButtonBase (Click)
Menuitem (Click)
Hyperlink (Click)
ListBoxItem (Double Click)

UI Elements with built-in Command Source support

Command
CommandTarget
CommandParameter

1b

InputBinding

(v) Gesture

MouseBinding

KeyBinding

RoutedCommand:

CanExecute(o, t)
Execute(o, t)
InputGestures
Name
OwnerType
^CanExecuteChanged

Contains (1 to n)

RoutedUICommand:

Text

Built-In (Predefined) Commands:
ApplicationCommands
NavigationCommands
EditingCommands
ComponentCommands
MediaCommands

WPF Commanding Summary

1b

InputGesture (abstract)

Matches(..)

MouseGesture

KeyGesture

Modifers
MouseAction
Matches(…)

DisplayString
Key
Modifiers
Matches(…)

Gesture
MouseAction

Gesture
Key
Modifiers

NewCmdSrc.cs – define new command src
NewCmds.cs – define new commands
CmdHandlers.cs – command logics
CmdBinding.xaml – command binding in XAML

Adding Commanding to Custom Control (including UserControl)

Credit: Pro WPF in C# 2008 (Mathew McDonald)

// 1) Setup command bindings

```
CommandBinding cmdBinding = new CommandBinding(ApplicationCommands.Undo, Undo_Executed, Undo_CanExecute);
theCustomCtrl = CommandBindings.Add(cmdBinding);
```

// 2) RoutedEvent Handlers

```
private void Undo_Executed(object sender, ExecutedRoutedEventArgs e) {
   ... // do the work here
}

private void Undo_CanExecute(object sender, CanExecuteRoutedEventArgs e) {
   ... // setting bool to "e.CanExecute ="
}
```

// 3) Trigger the Undo Command

```
// a) Ctrl-Z when the Custom Control Element has the Focus
// b) Setup a Button to do Undo
<Button Command="Undo" CommandTarget="{Binding ElementName=<theCustomControlElement>}">Undo</Button>
```

==========

// (In Static Constructor) Adding Built-In Commanding (Class CommandBinding) to Custom Control

```
CommandManager.RegisterClassCommandBinding(typeof(<theCustomControlElementType>),
                                           new CommandBinding(ApplicationCommands.Undo, Undo_Executed, Undo_CanExecute));
```

Note: Undo_Execute() and Undo_CanExecute() must be static handler.

Note: this is similar to using EventManager.RegisterClassHandler() to built-in routed event handler to a Custom Control.

Templates

All Templates Have Triggers

DataTemplate

Assigned To

(ContentControl.**ContentTemplate** or ItemsControl.**ItemTemplate)**

As supposedly ToString(), DataTemplate is used to specify how to display non-visual data in ContentControl or ItemsControl.

Eg: (**MyPhoto** is a Collection of **Photo**)

```
<Window.Resources>
    ...
    <DataTemplate DataType="{x:Type local:Photo}">
        <Border Margin="3">
            <Image Source="{Binding Path=Source}"/>
        </Border>
    </DataTemplate>
</Window.Resources>

<ListBox
    ItemsSource="{Binding Source="{StaticResource MyPhotos}"
    Background="Silver" SelectionIndex="0"/>
```

Binding Source is

Note:

Because it has to do with Data, DataBinding is almost always in the mix.

If **DataType** is specified with a type while no x:Key set (actually it is implicitly set as x:Key="{x:Type …}", the DataTemplate is applied whenever the type appears.

DataTemplateSelector: (Select a DataTemplate based on Data's Property value)
(ContentControl.**ContentTemplateSelector**, ItemsControl.**ItemTemplateSelector**)

1) Derive a class from DataTemplateSelector, and override the SelectTemplate() method to return a Template to use.

```
public override DateTemplate SelectTemplate(object item, …)
```

2) Instantiate an instance of the derived class and give it a x:Key.
3) Set either **ContentTemplateSelector** or **ItemTemplateSelector** property by "{StaticResource <resourceKey>}"

DataTrigger: Trigger to Apply to Property Values

```
<DataTemplate.Triggers>
    <DataTrigger Binding="{Binding Path=TaskType}">
        <DataTrigger.Value>
            <local:TaskType>Home</local:TaskType>
        </DataTrigger.Value>
            <Setter TargetName="border" Property="BorderBrush"
                                                Value="Yellow"/>
    </DataTrigger>
</DataTemplate.Triggers>
```

Note: Using Converter is at times a more efficient alternative.

What Belongs in a DataTemplate? (vs a Style)

In general, DataTemplate is concerned with only the presentation and appearance of the data objects; in most cases, all other aspects of presentation, such as what an item looks like when it is selected, does not belong in the definition of a DataTemplate. Those other aspects are mostly better served by using a Style.

ControlTemplate

Assigned To

(Control.**Template** or Page.**Template)**

Each Control has both Appearance and Behaviors.

Control's "Template" property reference to a ControlTemplate object which determined the appearance of the Control.

Control's behaviors is determined by control's Class object.

Eg:

```
<Style TargetType="Button">
    <Setter Property="Template">
        <Setter.Value>
            <ControlTemplate TargetType="Button">
                <Grid>
                    <Ellipse Fill="{TemplateBinding Background}"/>
                    <ContentPresenter/>
                </Grid>
            </ControlTemplate>
        </Setter.Value>
    </Setter>
</Style>
```

Note:

No notion of implicit key, even when TargetType is specified. (Unlike Style, DataTemplate), so in order to automatically apply a particular ControlTemplate, put it inside a Setter of a Style with desired implicit key:

```
<Style TargetType="{x:Type Button}">
    <Setter Property="Template" Value="{StaticResource btnControlTemplate}"/>
</Style>
```

TargetType is required if ContentPresenter, specifying where content should go, is used in ControlTemplate.

How to Specify where Control's Content go?

For ItemsControl:

Panel.**IsItemsHost** property, but ItemsPanel loses its function. **ItemsPresenter**, then use ItemsPanel (ItemsPanelTemplate) to control how Items is layout.

For ContentControl:

ContentPresenter

TemplateBinding: (Markup Extension)

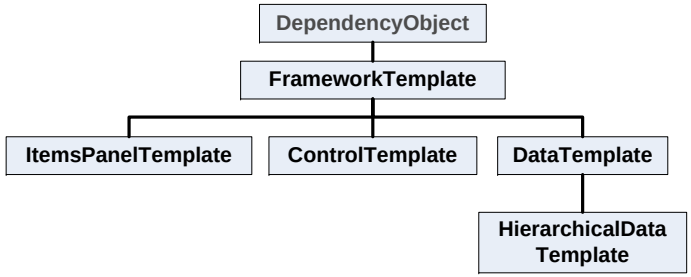
Control's already have many Properties defined; if ControlTemplate needs to use those Properties, then use TemplateBinding to refer them. Like the example above----Control's Background Property is used to set Ellipse's Fill Property in the ControlTemplate.

Note:

TemplateBinding can be done using regular DataBinding:

```
<Ellipse Fill="{Binding RelativeSource={RelativeSource TemplateParent}, Path=Background}"/>
```

How many Templates are there?



ItemsPanelTemplate

Assigned To

(ItemsControl.**ItemsPanel)**

Similar to ControlTemplate; "ItemControl" sheet has an example of its use.

HirarchicalDataTemplate

It is type of DataTemplate that is applied to a collection which contains other collections. (MSDN --- Data Templating Overview--- has an example of its user.

The Parallels Among Dependency Properpties and Routed Events

Dependency Property

A Dependency Property is set /stored on an object that defines the it, and has CLR wrapper.

In XAML: <Button Background="Red" x:Name="btn"/>
In Code: btn.Background = "Red";

Example: **Defining an Xyz Dependency Property of type "int"**

```
public class XyzOwner : DependencyObject { // Must be an DependencyObject to have D.Property

    public static readonly DependencyProperty XyzProperty = DependencyProperty.Register(
        "Xyz",                  // Property name
        typeof(int),            // Property value type
        typeof(XyzOwner), // Property Owner
        md);                    // Property metadata: Default value, PropertyChangeCallback, etc.

    public int Xyz              // CLR wrapper
    {
        get {return GetValue(XyzProperty);}
        set {SetValue(XyzProperty, value);}
    }
}
```

GetValue() and SetValue() are DependencyObject's Methods.

Different!

Routed Event

A Routed Event is set /stored on an object that defines it, and has CLR wrapper.

In XAML: <Button Click="Btn_Click" x:Name="btn"/>
In Code: btn.Click += Btn_Click;

Example: **Defining an Xyz Routed Event**

```
public class XyzOwner : UIElement { // Must be an UIElement or ContentElement to own a Routed Event

    public static readonly RoutedEvent XyzEvent = EventManager.RegisterRoutedEvent(
        "Xyz",                          // Event name
        RoutingStrategy.Bubble,         // Routing Strategy
        typeof(RoutedEventHandler),     // Event's Delegate Type
        Typeof(XyzOwner));              // Owner Type

    public event RoutedEventHandler Xyz         // CLR wrapper
    {
        add {AddHandler(XyzEvent, value);}
        remove {RemoveHandler(XyzEvent, value);}
    }
}
```

Same!

Attached Property

An Attached Property is Dependency Property, which is set/stored on an object other than the one that defines it, and has NO CLR wrapper. Attached Properties have an XAML syntax which must be supported by a coding pattern in the backing code.

In XAML:
```
<DockPanel>
    <Button DockPanel.Dock="Top" x:Name="btn"/>
</DockPanel>
```

In Code: **DockPanel**.Set**Dock**(btn, "Top");

Example: **Defining an Xyz Attached Property of type "int"**

```
public class XyzOwner : DependencyObject { // Must be an DependencyObject to have an Attached Property

    public static readonly DependencyProperty XyzProperty = DependencyProperty.RegisterAttached(
        "Xyz",                  // Property name
        typeof(int),            // Property value type
        typeof(XyzOwner), // Property Owner
        md);                    // Property metadata: Default value, PropertyChangeCallback, etc.

    public static int GetXyz(DependencyObject o)
    {
        return o.GetValue(XyzOwner.XyzProperty);
    }
    public static void SetXyz(DependencyObject o, value)
    {
        o.SetValue(XyzOwner.XyzProperty, value);
    }
}
```

GetXyz and SetXyz are Code Pattern reqired by XAML

Attached Event

An Attached Event enables adding event handler to ANY UIElement or ContentElement rather than to an element that actually defines or inherits the event, and it has NO CLR event wrapper. Attached events have an XAML syntax which must be supported by a coding pattern in the backing code.

In XAML:
```
<DocPanel Button.Click="Btn_Click" x:Name="dockp">
    <Button/>
</DockPanel>
```
In Code: **Button**.Add**Click**EventHandler(dockp, Btn_Click);

Example: **Defining an Xyz Routed Event**

```
public class XyzOwner : UIElement { // Must be an UIElement or ContentElement to own a Routed Event

    public static readonly RoutedEvent XyzEvent = EventManager.RegisterRoutedEvent(
        "Xyz",                          // Event name
        RoutingStrategy.Bubble,         // Routing Strategy
        typeof(RoutedEventHandler),     // Event's Delegate Type
        Typeof(XyzOwner));              // Owner Type

    public static void AddXyzHandler(UIElement u, RoutedEventHandler h)
    {
        u.AddHandler(XyzEvent, h);
    }
    public static void RemoveXyzHandler(UIElement u, RoutedEventHandler h)
    {
        u.RemoveHandler(XyzEvent, h);
    }
}
```

AddXyzHandler and RevmoveXyzHandler are Code Pattern reqired by XAML

Dependency Property vs Attached Property vs WPF Property System

WPF Property System is the backing for both Dependency Property and Attached Property---their values are store inside the Property System! To set (create) or retrieve a value from Property system, you need 1) a Property Id (same for both Dependency Property or Attached Property) gotten when register the Property (Dependency or Attached Property) and an instance of class---that is where the important difference between Dependency Property and Attached Property.

Aside from making different APIs to register Dependency and Attached Property, the most important difference between them is what instances are used to access the value----a Dependency Property use Owner Class' instance while an attached property uses instances where property is attached to. (How to come up with their associated values is a different story; in short, it depends☺)

Dependency Property Supports:

- Settable by a Style
- Support Data Binding
- Settable with a Dynamic Resource reference
- Inherit a property value automatically from a parent element in the element tree.
- Animatable
- Change Notifiable
- Support Overridable Metadata---default value, value validation and coercion,
- Designer support

References: Illustrated WPF (Ch7 Dependency Property); MSDN (Custom Dependency Properties)

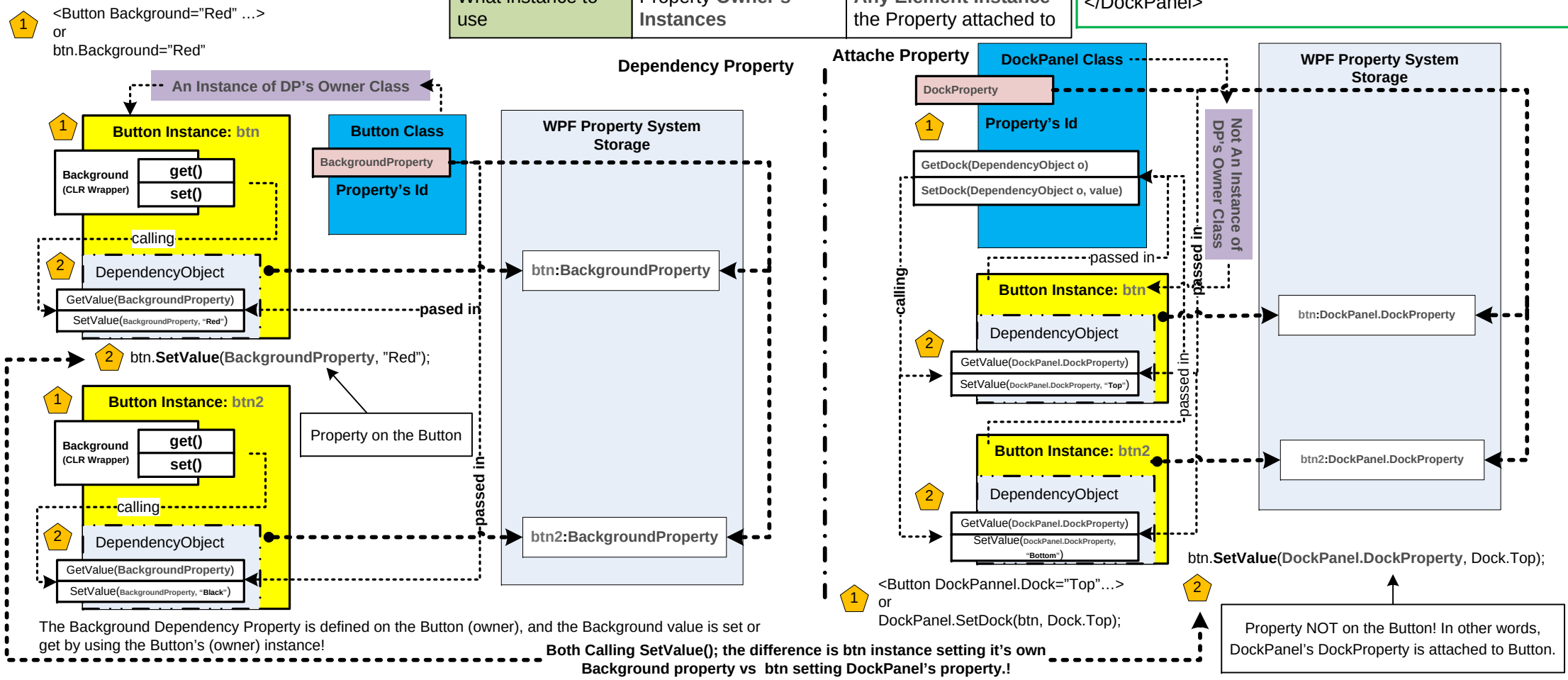
Property System's Identifications	Dependency Property	Attached Property
How to obtain unique Property Id:	DependencyProperty .Register()	DependencyProperty .RegisterAttached()
What instance to use	Property **Owner's Instances**	**Any Element Instance** the Property attached to

Example: **Dependency Property**

```
<Button x:Name="btn" Background="Red"/>
<Button x:Name="btn2" Background="Black"/>
```

Example: **Attached Property**

```
<DockPanel>
    <Button x:Name="btn" DockPanel.Dock="Top"/>
    <Button x:Name="btn2" DockPanel.Dock="Bottom"/>
</DockPanel>
```



Routed, Attached Event

Listener:
It is where Routed Event Handlers are added. Any UIElement or ContentElement can be listener for ANY Routed Event.

Source:
Where Routed Events are raised.

Owner:
Where Routed Events are defined.

RoutedEventHandler:
Basic routed event handler delegate
public delegate void RoutedEventHandler (object sender, RoutedEventArgs e);
sender is really the listener where handler is added.
e is the event data, RoutedEventArgs, the common event data base class.

Naming Convention: (For an Xyz Routed Event)
Id: XyzEvent
Delegate: XyzEventHandler
Data: XyzEventArgs

Adding Handler:

Instance Handler:

In XAML: <Button Click="On_Click" x:Name="btn"/>
In Code: btn.Click += new RoutedEventHandler(On_Click);
In Code: btn.AddHandler(Button.ClickEvent, new RoutedEventHandler(On_Click));

Class Handler: (Or overriding the OnXyz() on the base Elements---UIElement, ContentElement…**)**

EventManager.RegisterClassHandler
(Button, ClickEvent, new RoutedEventHandler(On_Click_Class));

Handled:
e.Handled = true;

HandledEventsToo:
If set to true, listening Handler will be invoked.
Setting handledEventsToo:
In Code: AddHandler(XyzEvent, Delegate, **true**); // Recommended way.
In XAML: <EventSetter Event=Click Handler="On_Click" **HandledEventsToo**=true/>

Class Handling:

A DependencyObject derived class can define and attach a class handler to a Routed Event that is a declared or inherited event member of your class. **Class handler are invoked before any instance listener handlers** that are attached to an instance of that class, whenever a Routed Event reaches an element instance in it route.

Some WPF controls have inherent class handling for certain routed events. This might give the outward appearance that the routed event is not ever raised, but in reality it is being class handled, and the routed event can potentially still be handled by instance handlers if certain techniques are used. Also, many base classes and controls expose virtual methods that can be used to override class handling behavior.

Attached Event: (Used in XAML when Event Owner will never be the Event Listener)
An attached event enables you to add a handler for a particular event to an arbitrary element (UIElement/ ContentElement). The element handling the event need not define or inherit the attached event, and neither the object potentially raising the event nor the destination handling instance must define or otherwise "own" that event as a class member.

Owning Aspect: (Owner who Registers the Routed Event)

Raising Aspect: (Source on which element the Routed Event is raised)
[src.RaiseEvent(Event, new RoutedEventArg(….))]

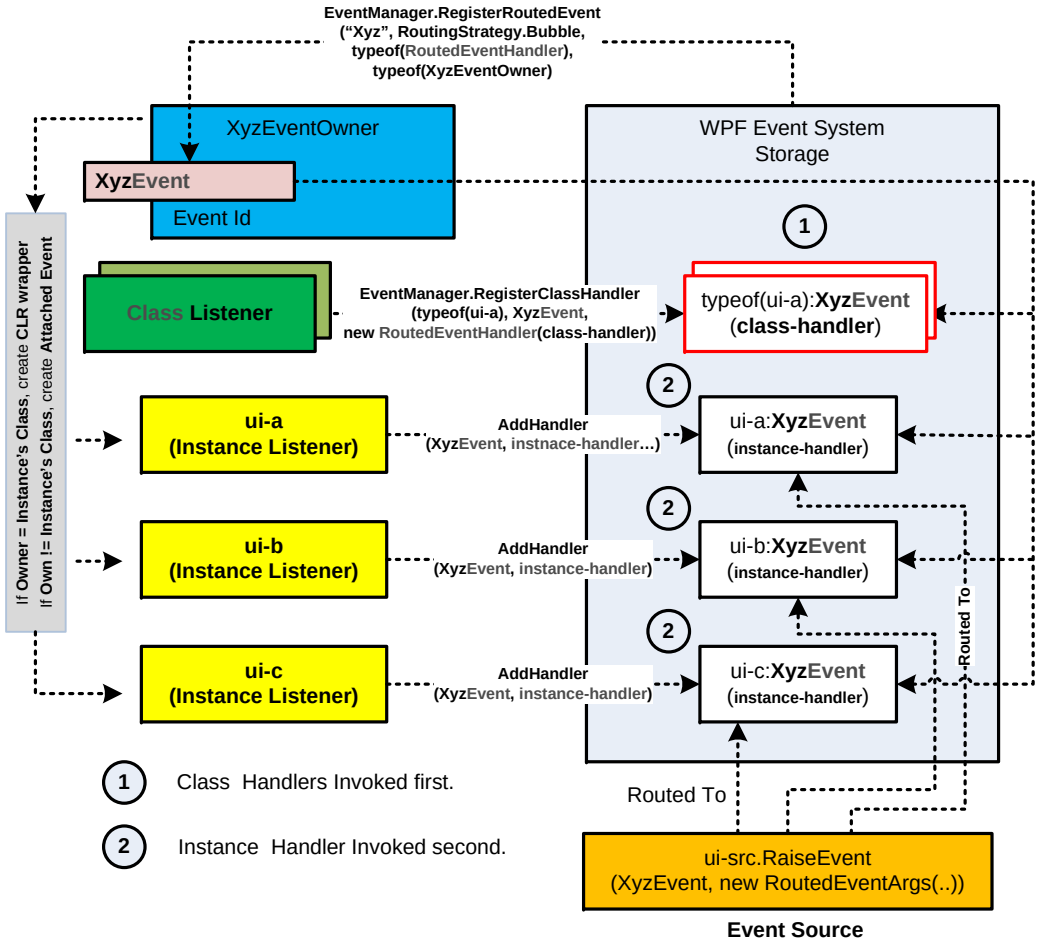
Routing Aspect:
Tunneling, Bubbling, and Direct---which are determined at the time of Routed Event registration.

Handler Aspect: (at a Listener, which is an UI instance, mostly)

Instance Handler:
uiInstance.AddHandler(XyzOwner.XyzEvent, new RoutedEventHandler(handler));

Class Handler:
EventManager.RegisterClassHandler
(typeof(uiInstance), XyzOwner.XyzEvent, new RoutedEventHanler(handler_);

Virtual Convention:
protect virtual void **On**Xyz(RoutedEventArgs e); // Meant to be overriden to do Class Handling



Credit: Essential Windows Presentation Foundation (Chris Anderson)

WPF Threading

In the World of Windows, MultiThread has **TWO** areas of importance:

1) **Synchronizations** between Threads---this is universal for any MultiThreaded implementation, include the Unix World.
2) (This is Windows Specific) **Marshalling** between different Thread Apartments (both STA and MTA), and Marshalling between a Thread Apartment (STA or MTA) and a Free Thread (System.Threading.Thread---aka WorkerThread, or ThreadPool Thread).

Objects created in a Thread Apartment (both a STA or a MTA) are only accessible by Threads in that Apartment! This is very different from the Shared Memory Threading Model that used by Unix. Thus, in order for an outside Thread (including STA, MTA, and Free Thread) to access an object in STA or MTA, that outside Thread need to ask the Thread(s) in the STA or MTA to do the access on its behalf. That's Marshalling!

UI Thread's Affinity: In WPF (WinForm, too), the main UI Thread uses the Single Threaded Apartment (STA) Threading Model; that means all UI elements are owned by the UI Thread, the single thread in the STA. In WPF, there is a Dispatcher created to manage that single UI Thread. To get current Dispatcher associated with the current UI Thread, use the static Dispatcher.CurrentDispatcher property within the UI Thread.

Cases of Threading: (In the WPF context)

Case 1: Only One Thread---the UI Thread.
Case 2: Use a Worker Thread. For a Worker Thread, created to do async operations, to interact with the UI Thread (returns results or does works in UI Thread), it needs to use UI Thread's Dispatcher.BeginInvoke() or Invoke() to queue a work-item to the Dispatcher's Priority Work Queue.
Case 3: (**Async Programming**) Use BackgroundWorker, which internally uses Asynchronous Delegate which uses a Thread from ThreadPool. The UI Thread creates the BackgroundWorkder; they communicate with each other through events. Events---ProgressChanged, RunWorkderCompleted---are raised in UI Thread, while DoWork event is raised in the BackgroundWorker. (Uses SynchronizationContext behind the scene)
Case 4: (Sync and Async Programming) Use SynchronizationContext directly. In UI thread, get its SyncrhronizationContext, by SynchronizationContext.Current, then in the Worker Thread or ThreadPool Thread, send or post back the results to UI Thread, by using the SynchronizationContext.

```
private void button1_Click(object sender, EventArgs e)
{
  ctx = SynchronizationContext.Current;
  ThreadPool.QueueUserWorkItem(
     delegate(object state) {
       // would be true---executing on ThreadPool Thread
       Debug.WriteLine(Thread.CurrentThread.IsThreadPoolThread.ToString());
       ctx.Send(delegate(object someState) {
         // would be false---executing on UI Thread
         label1.Text = Thread.CurrentThread.IsThreadPoolThread.ToString();
         }, null);
     });
}
```

4A. (Sync Programming)

3. BackgroundWorker (System.ComponentModel.BackgroundWorker)

= Provide easy model for doing Async operation; no explicit synchronization code to be written
= Good for run-to-finish scenarios; has progress reporting and cancellation capabilities

```
...
xmlns:cm="clr-namespace:System.ComponentModel;assembly=System"
…>
<Window.Resources>
    <cm:BackgroundWorker x:Key="bgWorker"
        WorkerReportsProgress = "true"
        WorkerSupportsCancellation = "true"
        DoWork = "bgw_DoWork"
        ProgressChanged = "bgw_ProgressChanged"
        RunWorkerCompleted = "bgw_RunWorkerCompleted"/>
</Window.Resources>
```

Note:
DoWork event is raised in the BackgroundWorker Thread.
ProgressChanged and **RunWorkerCompleted** events are raised in the UI Thread

```
// Get the BackgroundWorker instance from Resources
backgroundWorker bgw = (BackgroundWorker)this.FindResource("bgWorker");
...
// In the UI thread, to kick off the BackgroundWorker Thread
bgw.RunWorkerAsync(input);  // raise the DoWork event in the UI thread
```

In the BackgroundWorker thread, inside bgw_DoWork(object sender, DoWorkEventArgs e):

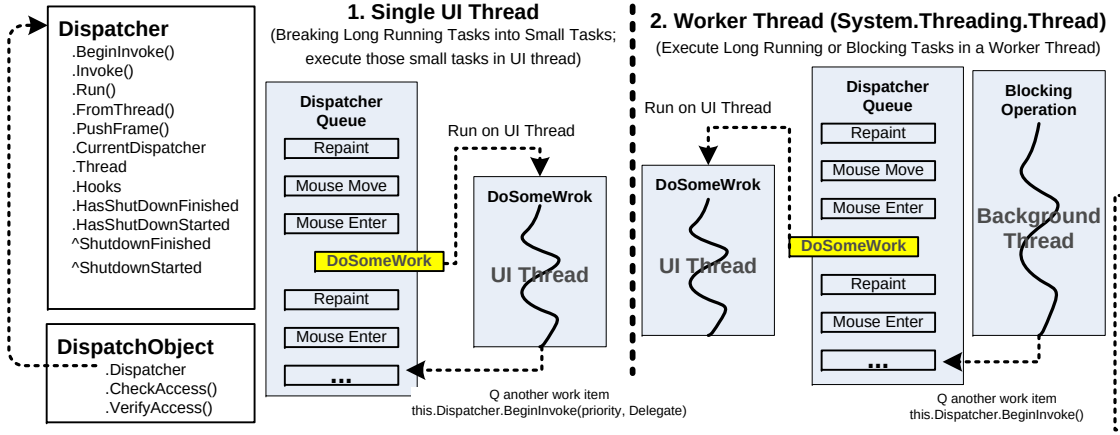
e.Argument, to get the argument, cast to the right object type.
e.Result, to return result back to the UI thread.

bgw.ReportProgress(<%>), to raise the ProgressChanged event on UI thread.

(UI thread's bgw_ProgressChanged(object sender, ProgressChangedEventArgs e) uses e.ProgressPercentage to get the percentage information)

In UI thread, use bgw.CancellAsync() to initiate cancellation of the async operation.
(In BackgroundWorker thread, bgw_DoWork(object sender, DoWorkEventArgs e) needs to poll the e.CancellationPending, like:

If (bgw.CanellationPending)… and send the e.Cancel = true; before returning.



4. (Async Programming, Using SynchronizationContext)

1) Define the **Result** object (message)

```
class SumCompletedEventArgs: EventArgs {
    public int Result { get; set; }
}
```

3) Using it:

```
void RunTask(object s, RoutedEventArgs e) {
    Sum t = new Sum();
    t.SumCompleted +=
    delegate(object s2, SumCompletedEventArgs e2) {
        _text.Text = e2.Result.ToString();
    };
    t.SumAsync(250);
}
```

Note: BackgoundTask() calls **OnCompleted**() to raise an event on the UI Thread, but an event doesn't has to be used; however, an event pattern does make it flexible, so that **RunTask**() can attach any piece of code to handle the result. Also, **SumAsync**() can use **ThreadPool.QueueUserWorkerItem**() instead of Thread().

2) Define the **Task** object to do the Async Operations:

```
class Sum {  // Task object
    SyncrhonizationContext _ctx;
    public Sum() {_ctx = SynchronizationContext.Current;}

    public event EventHandler<SumCompleteedEventArgs> SumCompleted;
    public void SumAsync(int value) { // UI Thread to spawn the Worker Thread
        Thread background=new Thread(BackgroundTask);
        background IsBackground = true;
        background.Start(value);
    }
    void BackgroundTask(object p) { // Executed in the Worker Thread
        Int value = (int)p; int result=0;
        //...Calculate the result
        _ctx.Post(OnCompleted, result); // To call back to UI Thread (Async)
    }
    void OnCompleted(object result) { // raise SumCompleted on UI Thread
        if (SumComplete != null) {
            SumCompletedEventArgs e = new SumCompletedEventArgs();
            e.Result = (int) result;
            SumCompleted(this, e); // Raise SumCompleted event
        }
    }
}
```

Event-Based Async Programming Model (APM), Using SynchronizationContext

1) Define the **Result** object (message)

```
class SumCompletedEventArgs: EventArgs {

    public int Result { get; set; }
}
```

3) Using the Task Object:

```
void RunTask(object s, RoutedEventArgs e) {

  Sum t = new Sum();

  t.SumCompleted +=  delegate(object s2, SumCompletedEventArgs e2) {
    _text.Text = e2.Result.ToString();
  };

  t.SumAsync(250); // calls the Async method
}
```

Raises the Event

Event Raised on UI Thread: BackgoundTask() calls OnCompleted() to raise an event on the UI Thread, but an event doesn't has to be used, because the OnCompleted() is run on the UI thread already; however, an event pattern does make it flexible, so that **RunTask**() can attach any piece of code to handle the result. Also, **SumAsync**() can use **ThreadPool.QueueUserWorkerItem**() instead of Thread().

Progress Report: Like using the SumCompleted event, we can create ProgressChanged event to report Progress and Incremental Results in the event arguments (need to be defined in the same fashion as the SumCompletedEventArgs).

Cancel Async Op: Task object (Sum) can easily support Cancellation by implementing a CancelAsync() method.

Concurrent Invocations: To support concurrent invocation, SunAsync() needs a 2nd parameter to uniquely identify a running task.

2) Define the **Task** object that supports Event-Based Async Operations:

```
class Sum {  // Task object

  SyncrhonizationContext _ctx;
  public Sum() {_ctx = SynchronizationContext.Current;} // save syncCtx

  public event EventHandler<SumCompleteedEventArgs> SumCompleted;
  public void SumAsync(int value) { // UI Thread to spawn the Worker Thread

    Thread background=new Thread(BackgroundTask);
    background IsBackground = true;
    background.Start(value);
  }

  void BackgroundTask(object p) { // Executed in the Worker Thread

    int value = (int)p; int result=0;
    //...Calculate the result
    _ctx.Post(OnCompleted, result); // To call back to UI Thread (Async)
  }

  void OnCompleted(object result) { // Raises SumCompleted on UI Thread

    if (SumComplete != null) {

      SumCompletedEventArgs e = new SumCompletedEventArgs();
      e.Result = (int) result;
      SumCompleted(this, e); // Raise SumCompleted event
    }
  }
```

Credit: Essential WPF by Chris Anderson (pp425)

GUI and Threadings

Good for Application Logic
tied to UI Thread

UI Thread

disp = Dispatcher.CurrentDispatcher()

Credit: http://msdn.microsoft.com/en-us/library/ms741870.aspx (Threading Model)

Thread or ThreadPool Thread

System.Threading.Thread()
ThreadPool Thread

disp.BeginInvoke(Delegate, Priority, params object[])

UI Thread

```
var bgw = new BackgroundWorker()
bgw.RunWorkerAsync()
bgw.CannelAsync();
```

Raises

Events:
ProgressChanged
RunWorkerCompleted

Raises

Raises
Note: The DoWork event
handler does not need to call
OnRunWorkerCompleted();
it just need to return();

BackgroundWorker

Event:
DoWork

ReportProgress()
OnRunWorkerCompleted()

Async Delegate

IAsyncResult = d.BeginInvoke() [Non-block]
d.EndInvoke() [Block, to Get Results]

IAsyncResult
.IsCompleted (for Polling)
.WaitHandle (to WaitOne())

AsyncCallback

Delegate
To Pass Into

ThreadPool
.QueueUserWorkItem()

UI Thread

```
var t = new Task();
t.TaskCompleted += TaskEventHandler;
t.TaskAsync(205);

void TaskEventHandler(object s, TaskCompletedEventArgs e) {
      // does the UI Works!!!!!
}
```

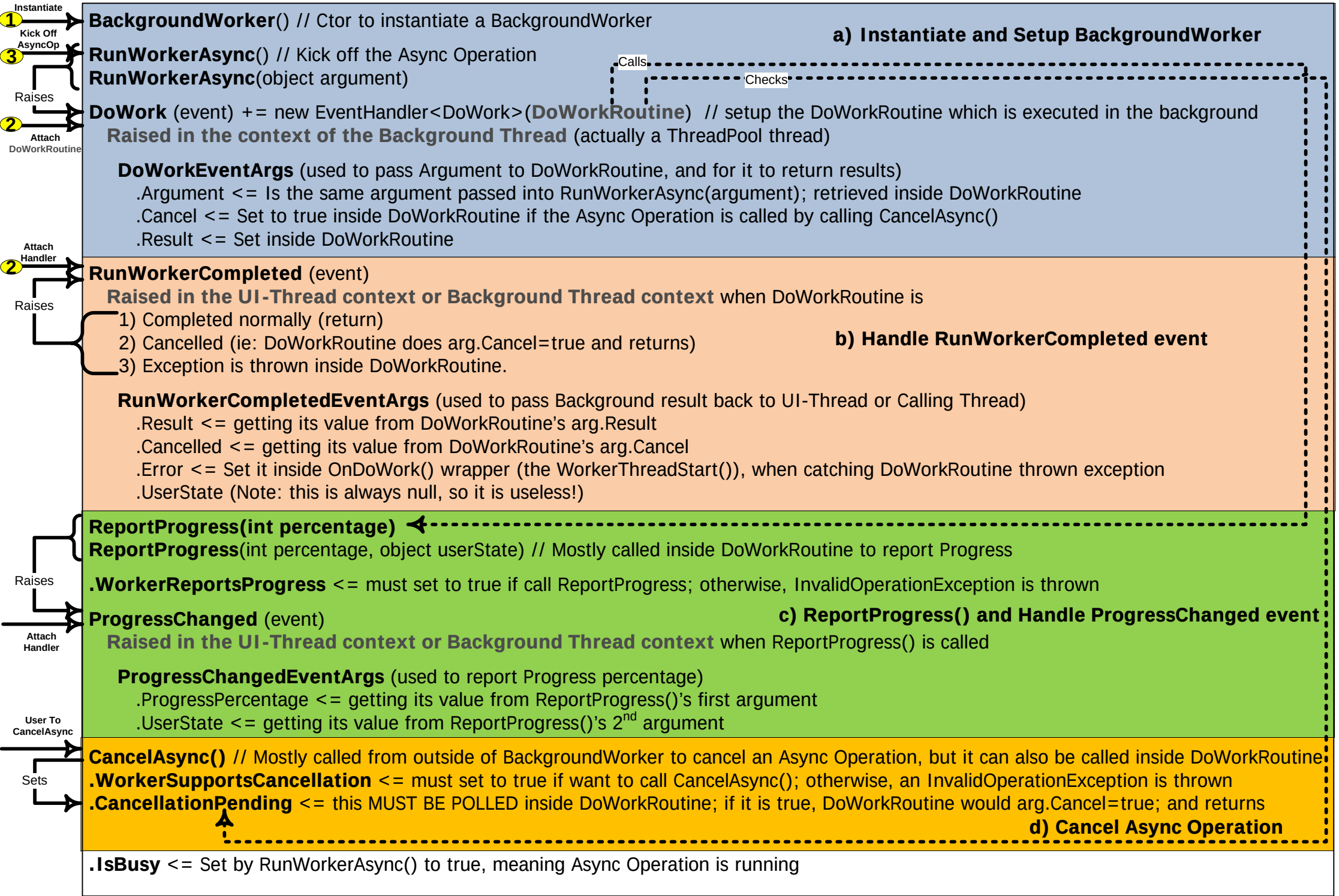
Good for reusable Component

```
Task() { this._ctx = SyncrhonizationContext.Current;}
public event EventHandler<TaskCompletedEventArgs> TaskCompleted;
void TaskAsync(int value) {
      System.Threading.Thread(new ThreadStart(BgTask) or ThreadPool Thread
}
void BgTask(object p) {
      // .. Do work here!
      this._ctx.Post(OnTaskCompleted, result);
}
void OnTaskCompleted(object result) {
      //...some checking 1st
      TaskCompleted(this, e);
}
```

Credit: Essential WPF Chris Anderson (Appendix)

BackgroundWorker (Internals)

Credit: Reflector on BackgroundWorker class



WPF Drawing, Visual, and Geometry vs Brushes

XAML Element Containment
Class Hierarchy
<Class Instance in XAML>

Path
.Stroke
.Fill
StrokeThickness

draws the shape described by <Geometry> classes, and is an UIElement.

<Path>
<Path.Data>

Path.Data

defines an abstract shape or a path, can be applied to brushes and UIElement's Clip Property, and is a Freezable; uses Geometry Mini-Lang.

<Geometry>
.Transfrom

- LineGeometry
- RectangleGeometry
- EllipseGeometry

LineGeometry StartPoint = "10,50" EndPoint = "100,150"/>

GeometryGroup
.FillRules

combing shapes and often used as a resource.
EvenOdd, NonZero

```
<Windows.Resources>
    <GeometryGroup x:Key="Geometry">
        <RectangleGeometry Rect="0,0 100,100"/>
        <EllipseGeometry Center="150,50" RadiusX="35" RadiusY="25"/>
    </GeometryGroup>
</Windows.Resources>

<Canvas>
    <Path Fill="Yellow" Stroke="Blue" Margin="5" Canvas.Top="10" Canvas.Left="10"
    Data ="{StaticResource Geometry}"/>
</Canvas>
```

CombineGeometry
.Geometry1
.Geometry2
.GeometryCombineMode

Union, Intercept, Xor, Exclude

```
<CombinedGeometry GeometryCombineMode="Union">
    <CombinedGeometry.Geometry1>
        <RectangleGeometry Rect="0,0 100,100"/>
    </CombinedGeometry.Geometry1>
    <CombinedGeometry.Geometry2>
        <EllipseGeometry Center="85,50" RadiusX="65" RadiusY="35"/>
    </CombinedGeometry2>
</CombinedGeometry>
```

simple, readonly PathGeometry

- StreamGeometry
- PathGeometry

- Brush
  - TileBrush
    - DrawingBrush .Draw → DrawingImage
    - ImageBrush .ImageSource → DrawingImage
    - VisualBrush .Visual → DrawingVisual

```
<Button …>
    <Image …>
        <Image.Source>
            <DrawingImage>
                <DrawingImage.Drawing>
                    <DrawingGroup>
                        <GeometryDrawing…>
                        <GeometryDrawing…>
                        <GeometryDrawing…>
                    </DrawingGroup>
                </DrawingImage.Drawing>
            </DrawingImage>
        </Image.Source>
    </Image>
</Button>
```

Path — <Path Stroke="Blue">
Path.Data — <Path.Data>
PathGeometry — <PathGeometry>
PathGeometry.Figures (Content Property)

PathFigure
.StartPoint
.IsClosed
.IsFilled

PathFigure.Segments (Content Property)

<PathSegment>

- LineSegment
- ArcSegment
- BezierSegment
- QuadraticSegment
- PolyBezierSegment
- PolylineSegment
- PolyQuadraticBezierSegment

```
In XAML:
<Path Stroke="Blue">
    <Path.Data>
        <PathGeometry>
            <PathFigure IsClosed="True" StartingPoint="10,100">
                <LineSegment Point="100,100"/>
                <LineSegment Point="100,50"/>
            </PathFigure>
        </PathGeometry>
    </Path.Data>
</Path>
```

```
<PathFigure IsClosed="True" StartingPoint="10,100">
...
```

In Code-Behind (3 Steps):

1. pathFigure = new PathFigure(); pathFigure.IsClosed=false; pathFigure.StartPoint=pt;
   pathFigure.Segments = new PathSegmentCollection();
   path.Figure.Segments.Add(new LineSegment(pt, true));
2. pathGeometry = new PathGeometry();
   pathGeometry.Figures = new PathFigureCollection();
   pathGeometry.Figures.Add(pathFigure);
3. path = new Path(); path.StrokeThickness = 3.0f; path.Stroke="red";
   path.Data = pathGeometry;

```
<LineSegment Point="100,100"/>
<LineSegment Point="100,50"/>
```

<Drawing> represents a 2-D drawing, containing all info need to display a piece of vector or bitmap art; they are not elements and can't be placed in user interfaces!

GeometryDrawing
.Geometry
.Brush
.Pen

ImageDrawing
.ImageSource
.Rect

VideoDrawing
.Player
.Rect

GlyphRunDrawing

DrawingGroup
.BitmapEffect
.BitmapEffectInput
.Children
.ClipGeometry
.GuidelineSet
.Opacity
.OpacityMask
.Transform

```
<GeometryDrawing Brush="Yellow">
    <GeometryDrawing.Pen>
        <Pen Brush="Blue" Thickness="3"/>
    </GeometryDrawing.Pen>
    <GeometryDrawing.Geometry>
        <PathGeometry>
            <PathFigure IsClosed="True" StartingPoint="10,100">
                <LineSegment Point="10,100"/>
                <LineSegment Point="100,50"/>
            </PathFigure>
        </PathGeometry>
    </GeometryDrawing.Geometry>
</GeometryDrawing>
```

Path vs PathGeometry vs PathFigure vs PathSegment

Path (PathData) has ONE PathGeometry;
One PathGeometry (PathGeometry.Figures) has MANY PathFigures
One PathFigure (PathFigure.Segments) has MANY PathSegments

Brushes and Their Applications

SolidColorBrush
.Color

```
<Rectangle Width=”75" Height=”75">
    <Rectangle.Fill>
        <SolidColorBrush Color=”Red”/>
    </Rectangle.Fill>
</Rectangle>
```

ImageBrush
.ImageSource

```
<Rectangle Width=”75" Height=”75">
    <Rectangle.Fill>
        <ImageBrush ImageSource=”mypic.jpg”/>
    </Rectangle.Fill>
</Rectangle>
```

RadialGradientBrush
.GradientStop

```
<Rectangle Width=”75" Height=”75">
    <Rectangle.Fill>
        <RadialGradientBrush GradientOrigin=”0.75,0.25">
            <GradientStop Color=”Yello” Offset=”0.0"/>
            <GradientStop Color=”Orange” Offset=”0.5"/>
            <GradientStop Color=”Red” Offset=”1.0"/>
        </RadialGradientBrush>
    </Rectangle.Fill>
</Rectangle>
```

Panel
.Background

Pen
.Brush

Control
.Background
.Foreground

Textblock
.Background

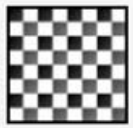
Border
.BorderBrush

Shape
.Fill
.Stroke

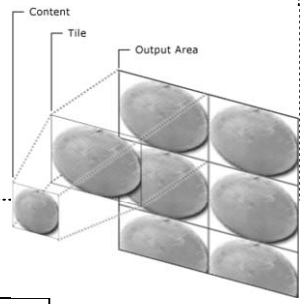
LinearGradientBrush
.GradientStop

```
<Rectangle Width=”75" Height=”75">
    <Rectangle.Fill>
        <LinearGradientBrush>
            <GradientStop Color=”Yello” Offset=”0.0"/>
            <GradientStop Color=”Orange” Offset=”0.5"/>
            <GradientStop Color=”Red” Offset=”1.0"/>
        </LinearGrandientBrush>
    </Rectangle.Fill>
</Rectangle>
```

DrawingBrush
.Drawing

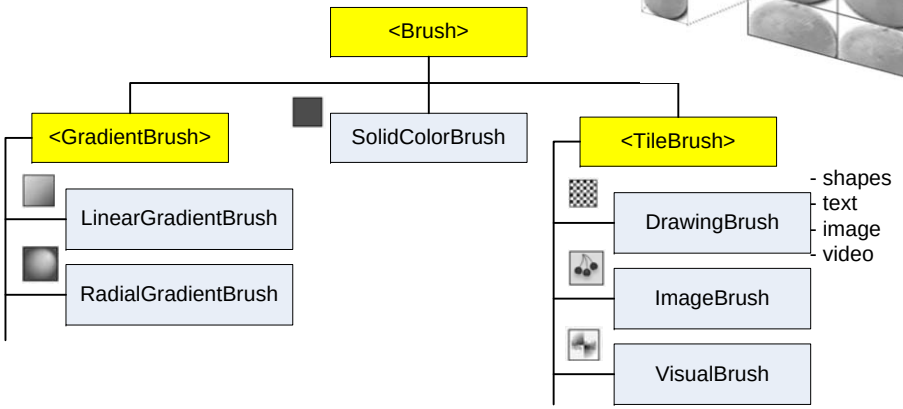


```
<Rectangle Width=”75" Height=”75">
    <Rectangle.Fill>
        <DrawingBrush Viewport=”0,0,0.25,0.25" TileMode=”Tile”>
            <DrawingBrush.Drawing>
                <DrawingGroup>
                    <GeometryDrawing Brush=”White”>
                        <GeometryDrawing.Geometry>
                            <RectangleGeometry Rect=”0,0,100,100"/>
                        </GeometryDrawing.Geometry>
                    </GeometryDrawing>

                    <GeometryDrawing>
                        <GeometryDrawing.Geometry>
                            <GeometryGroup>
                                <RectangleGeometry Rect=”0,0,50,50"/>
                                <RectangeGeometry Rect=”50,50,50,50"/>
                            </GeometryGroup>
                        </GeometryDrawing.Geometry>

                        <GeometryDrawing.Brush>
                            <LinearGradientBrush>
                                <GradientStop Offset=”0.0" Color=”Black”/>
                                <GradientStop Offset=”1.0" Color=”Gray”/>
                            </LinearGradientBrush>
                        </GeometryDrawing.Brush>
                    </GeometryDrawing>
                </DrawingGroup>
            </DrawingBrush.Drawing>
        </DrawingBrush>
    </Rectangle.Fill>
</Rectangle>
```

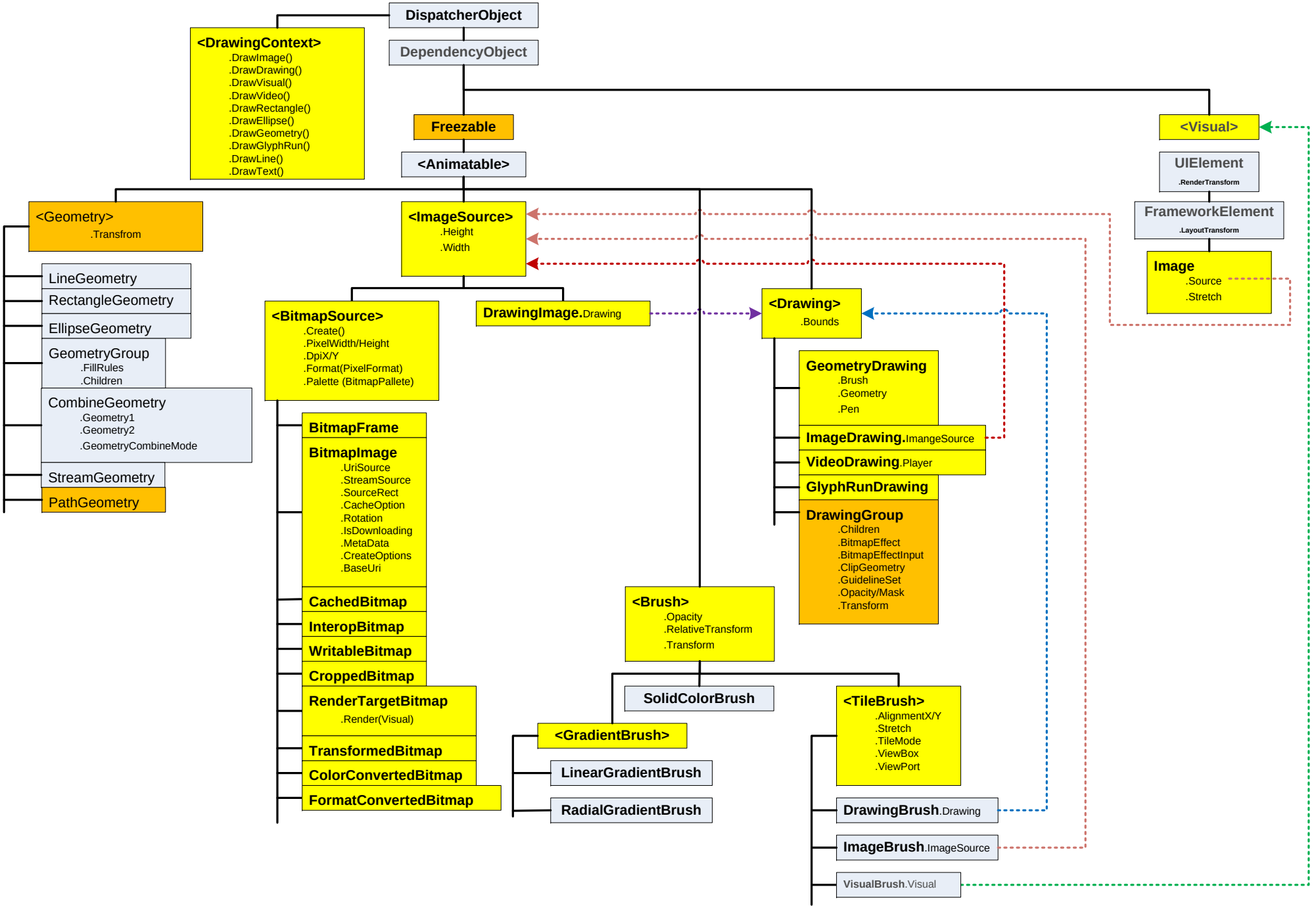


VisualBrush
.Visual


```
<Rectangle Width="75" Height="75">
    <Rectangle.Fill>
        <VisualBrush TileMode="Tile">
            <VisualBrush.Visual>
                <StackPanel>
                    <StackPanel.Background>
                        <DrawingBrush>
                            <DrawingBrush.Drawing>
                                <GeometryDrawing>
                                    <GeometryDrawing.Brush>
                                        <RadialGradientBrush>
                                            <GradientStop Color="MediumBlue" Offset="0.0" />
                                            <GradientStop Color="White" Offset="1.0" />
                                        </RadialGradientBrush>
                                    </GeometryDrawing.Brush>

                                    <GeometryDrawing.Geometry>
                                        <GeometryGroup>
                                            <RectangleGeometry Rect="0,0,50,50" />
                                            <RectangleGeometry Rect="50,50,50,50" />
                                        </GeometryGroup>
                                    </GeometryDrawing.Geometry>
                                </GeometryDrawing>
                            </DrawingBrush.Drawing>
                        </DrawingBrush>
                    </StackPanel.Background>
                    <TextBlock FontSize="10pt" Margin="10">Hello, World!</TextBlock>
                </StackPanel>
            </VisualBrush.Visual>
        </VisualBrush>
    </Rectangle.Fill>
</Rectangle>
```



Bitmap, Brush, Geometry and Drawing



Styling and Templating an ItemsControl (From MSDN --- Data Templating Overview) Note: Property in **Red**, Templates in **Black**.

Even though the ItemsControl is not the only control type that you can use a DataTemplate with, it is a very common scenario to bind an ItemsControl to a collection. In general, the definition of your DataTemplate should only be concerned with the presentation of data. In order to know when it is not suitable to use a DataTemplate it is important to understand the different style and template properties provided by the ItemsControl. The following example is designed to illustrate the function of each of these properties---**Template** (for **ControlTemplate**), **ItemsPanel** (for **ItemsPanelTemplate**), and **ItemTemplate** (for **DataTemplate**). The ItemsControl in this example is bound to the same Tasks collection. For demonstration purposes, the styles and templates in this example are all declared inline.

```
<ItemsControl Margin="10" ItemsSource="{Binding Source={StaticResource myTodoList}}"> // The ItemsControl is bound to myTodList Resources

 <!--The ItemsControl has no default visual appearance. Use the Template property to specify a ControlTemplate to define
     the appearance of an ItemsControl. The ItemsPresenter specifies where items should go and relies on the ItemsPanel's
     ItemsPanelTemplate to layout the items. If an ItemsPanelTemplate is not specified, the default StackPanel is used.-->

 <ItemsControl.Template>
  <ControlTemplate TargetType="ItemsControl">
   <Border BorderBrush="Aqua" BorderThickness="1" CornerRadius="15">
    <ItemsPresenter/>
   </Border>
  </ControlTemplate>
 </ItemsControl.Template>

 <!--Use the ItemsPanel property to specify an ItemsPanelTemplate that defines the panel that is used to hold the generated
     items. In other words, use this property if you want to affect how the items are laid out.-->

 <ItemsControl.ItemsPanel>
  <ItemsPanelTemplate>
   <WrapPanel />
  </ItemsPanelTemplate>
 </ItemsControl.ItemsPanel>

 <!--Use the ItemTemplate to set a DataTemplate to define the visualization of the data objects. This DataTemplate specifies
     that each data object appears with the Priority and TaskName on top of a silver ellipse.-->

 <ItemsControl.ItemTemplate>
  <DataTemplate>
   <DataTemplate.Resources>
    <Style TargetType="TextBlock">
     <Setter Property="FontSize" Value="18"/>
     <Setter Property="HorizontalAlignment" Value="Center"/>
    </Style>
   </DataTemplate.Resources>
   <Grid>
    <Ellipse Fill="Silver"/>
    <StackPanel>
     <TextBlock Margin="3,3,3,0" Text="{Binding Path=Priority}"/>
     <TextBlock Margin="3,0,3,7" Text="{Binding Path=TaskName}"/>
    </StackPanel>
   </Grid>
  </DataTemplate>
 </ItemsControl.ItemTemplate>

 <!--Use the ItemContainerStyle property to specify the appearance of the element that contains the data. This
     ItemContainerStyle gives each item container a margin and a width. There is also a trigger that sets a tooltip that shows
     the description of the data object when the mouse hovers over the item container.-->

 <ItemsControl.ItemContainerStyle>
  <Style>
   <Setter Property="Control.Width" Value="100"/>
   <Setter Property="Control.Margin" Value="5"/>
   <Style.Triggers>
    <Trigger Property="Control.IsMouseOver" Value="True">
     <Setter Property="Control.ToolTip"
          Value="{Binding RelativeSource={x:Static RelativeSource.Self}, Path=Content.Description}"/>
    </Trigger>
   </Style.Triggers>
  </Style>
 </ItemsControl.ItemContainerStyle>
</ItemsControl>
```

On the right is a screenshot of the example when it is rendered:

Note: instead of using the **ItemTemplate** (specifying a **DataTemplate**), you can use the **ItemTemplateSelector** (specifying an instance of **ItemTemplateSelector** derived class). Similarly, instead of using the **ItemContainerStyle**, you have the option to use the **ItemContainerStyleSelector**.

Two other style-related properties of the **ItemsControl** that are not shown here are **GroupStyle** and **GroupStyleSelector**.

System.Windows.Ctonrols.InkCanvas

System.Windows.Ink

System.Windows.Input

InkCollector

InkPicture

InkOverlay

DefaultDrawingAttributes

Stroke

EditingMode

DrawingAttributes

Stylus

InkCavas

StylusPoint

System.Windows.Controls:

InkCanvas

System.Windows.Ink:

ApplicationGesture
DrawingAttributeIds
DrawingAttributes
EllipseStylusShape
IncrementalHitTester
IncrementalLassoHitTester
RecognitionConfidence
RectangleStylusShape
Stroke
StrokeCollection
StylusShape
StylusTip

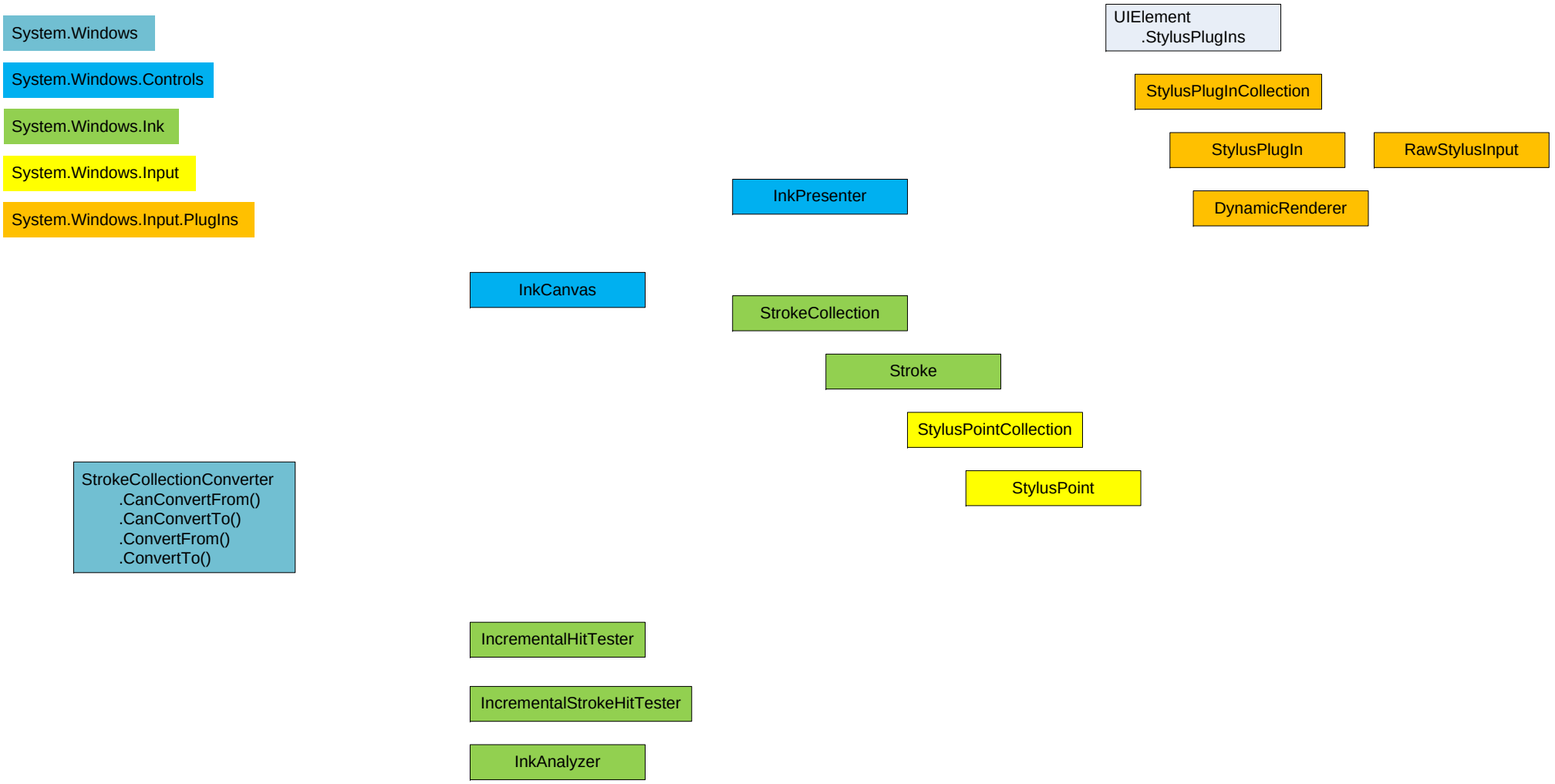
System.Windows.Input:

Stylus
StylusDevice
StylusDeviceCollection
StylusPoint
StylusPointCollection
StylusPointDescription
StylusPointProperties
StylusPointProperty
StylusPointPropertyInfo
StylusPointPropertyUnit

System.Windows.Input.
StylusPlugIns:

StylusPlugIn
DynamicRenderer
StylusPlugInCollection
RawStylusInput

WPF Ink


System.Windows
System.Windows.Controls
System.Windows.Ink
System.Windows.Input
System.Windows.Input.PlugIns
UIElement
.StylusPlugIns
StylusPlugInCollection
StylusPlugIn
RawStylusInput
DynamicRenderer
InkPresenter
InkCanvas
StrokeCollection
Stroke
StylusPointCollection
StylusPoint
StrokeCollectionConverter
.CanConvertFrom()
.CanConvertTo()
.ConvertFrom()
.ConvertTo()
IncrementalHitTester
IncrementalStrokeHitTester
InkAnalyzer

Layout

ArrangeOverride()

UIElementCollection
InternalChildren

Child.Arrange(rect)

Layout Transform vs RenderTransform

WPF Layout Steps:

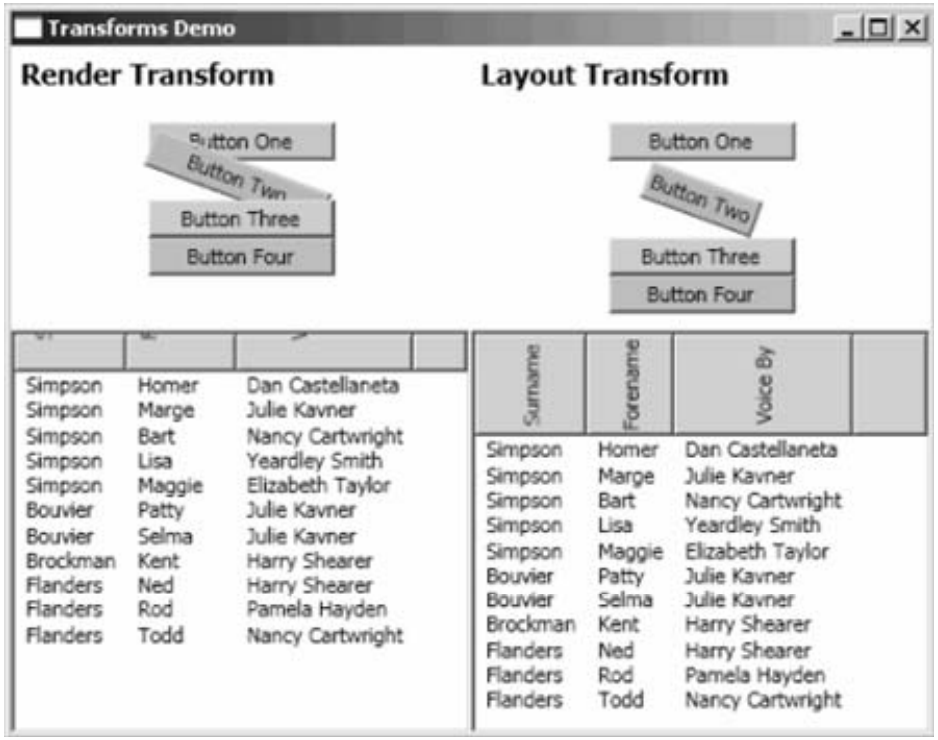
Measure
Arrange
Render

LayoutTransform and RenderTransform in Perspective:

LayoutTransform (before Layout pass)
Measure
Arrange
Render
} Layout Pass

RenderTransform (transform the final rendered output)

Note: In the Canvas layout container, both RenderTransform and LayoutTransform look behaving the same.



Animation

Moving the Scene:

Up and Down -- Mouse Wheel Scroll
Left and Right – Shift + Mouse Wheel Scroll
Center it – Double click the PAN tool
Spacebar – PAN tool
Maximize Scene -- TAB

Zooming the Scene:

In or Out -- Ctrl + Mouse Wheel Scroll
Actual Size – Double click the Zoom tool
Zoom Out – Alt + Zoom tool

Design and Animation Workspace Toggle:

F6

Disable Line Snapping Temporarily:

S

Access Tools by Key:

V – Selection
A – Direct Selection
H – Pan
SPC – Pan
Z – Zoom
O – Camera Orbit
I – Eyedropper
F – Paint Bucket
G – Brush Transform
P – Pen Tool
M – Rectangle
L – Ellipse
\ -- Line

Changing Property Value:

Increase – Click and Hold, then Drag Upward (Shift, faster; Control, slower)
Decrease – Click and Hold, then Drag Downward (Shift, faster; Control, slower)

Rotate by 15 degree:
Shift (pressed) while rotating

Exit Text Editing Mode:
Esc

Put a Square on the Scene:
Shift (while Dragging the Rectangle to the Scene)

Background with No Brush will not accept Click Events!!!!!

Creating a Brush resources from invisible Brush Source:

1) Group the Brush Source with a Container (like a Grid)
2) Make the Container invisible
3) Make sure the Brush Source is visible!

Make Window Borderless:
Set the AllowTransparency property to True

Handy Stuff:
Set CurrentSelection
Tool->MakeButton (to convert a selection to button)
Object->Make Motion Path
Object->Make Clipping Path

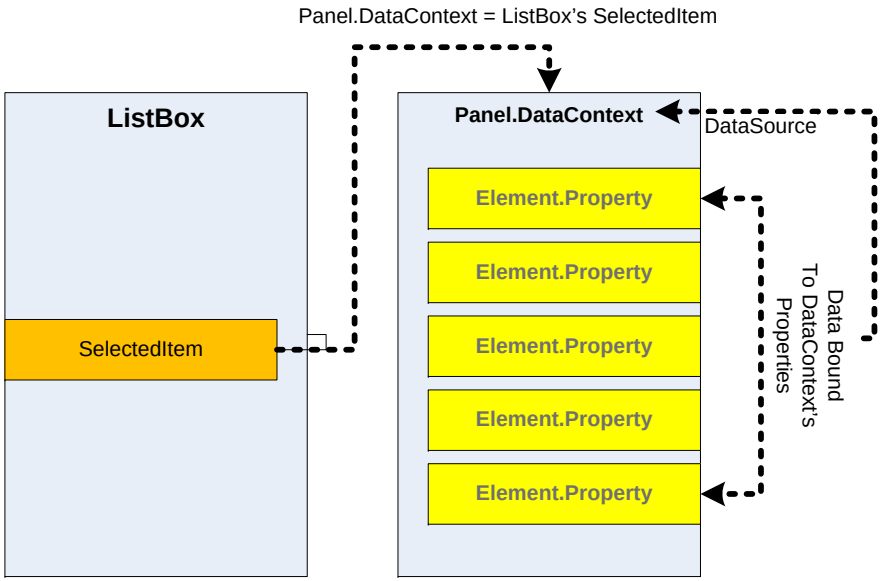
What is the Difference?

```
<Window.Resources>
    <ImageBrush x:Key="IcBackground" ImageSource="backache.jpg"/>
</Window.Resources>
      ...
<InkCanvas x:Name="ic" Background="{DynamicResource IcBackground}" .../>
                          vs
<InkCanvas x:Name="ic" Background="{Binding Source={StaticResource IcBackground}}" …/>
```

Can't use Dynamic, which only apply to Dependency Property.

What is the Difference?

```
<ImageBrush x:Key="myBG" ImageSource="my.jpg">

                vs

<ObjectDataProvider x:Key="myBG"
   ObjectType="{x:Type ImageBrush}"  MethodName="XYZ">
   <ObjectDataProvider.MethodParameters>
      …
   </ObjectDataProvider.MethodParameters>
</ObjectDataProvider>
```



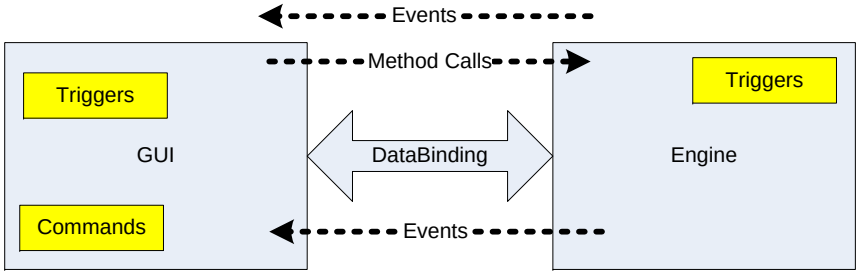
ObjectDataProvider

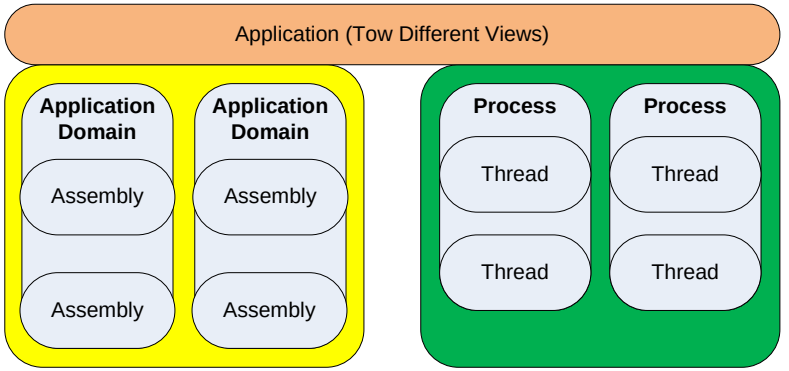
IsAsynchronous
MethodName, **Gets or sets the name of the method to call.**

MethodParameters
ObjectInstance
ObjectType, **Gets or sets the type of object to create an instance of.**

XmlDataProvider

BaseUri
Document
IsAsynchronous
Source
XmlNamespaceManager
XmlSerializer
XPath





x:Key

x:Name

x:Static

x:Array

{StaticResource… }

{DynamicResource… }

{Binding … }

Dispatcher things

The DispatcherTimer is reevaluated at the top of every Dispatcher loop.

Timers are not guaranteed to execute exactly when the time interval occurs, but are guaranteed to not execute before the time interval occurs. This is because DispatcherTimer operations are placed on the Dispatcher queue like other operations. When the DispatcherTimer operation executes is dependent on the other jobs in the queue and their priorities.

If a System.Timers.Timer is used in a WPF application, it is worth noting that the System.Timers.Timer runs on a different thread then the user interface (UI) thread. In order to access objects on the user interface (UI) thread, it is necessary to post the operation onto the Dispatcher of the user interface (UI) thread using Invoke or BeginInvoke. For an example of using a System.Timers.Timer, see the Disable Command Source Via System Timer sample. Reasons for using a DispatcherTimer opposed to a System.Timers.Timer are that the DispatcherTimer runs on the same thread as the Dispatcher and a DispatcherPriority can be set on the DispatcherTimer.

A DispatcherTimer will keep an object alive whenever the object's methods are bound to the timer.

So, the right way to schedule things inside the WPF UI is something like;
[csharp]
private DispatcherTimer _timer;
timer = new DispatcherTimer(DispatcherPriority.Background);
timer.Interval = TimeSpan.FromMinutes(5);
timer.Tick += delegate { ScheduleUpdate(); };
timer.Start();

[/csharp]
the timer is injected implicitly in the thread associated to the dispatcher of the UI.

DataModel

DataModel is responsible for exposing data in a way that is easily consumable by WPF. All of its public APIs must be called on the UI thread only. It must implement INotifyPropertyChanged and/or INotifyCollectionChanged as appropriate. When data is expensive to fetch, it abstracts away the expensive operations, never blocking the UI thread (that is evil!). It also keeps the data “live” and can be used to combine data from multiple sources. These sorts of classes are fairly straightforward to unit test.

ViewModel

A ViewModel is a model for a view in the application (duh!). It exposes data relevant to the view and exposes the behaviors for the views, usually with Commands. The model is fairly specific to a view in the application, but does not subclass from any WPF classes or make assumptions about the UI that will be bound to it. Since they are separate from the actual UI, these classes are also relatively straightforward to unit test.

View

A View is the actual UI behind a view in the application. The pattern we use is to set the DataContext of a view to its ViewModel. This makes it easy to get to the ViewModel through binding. It also matches the DataTemplate/Data pattern of WPF. Ideally, the view can be implemented purely as Xaml with no code behind. The attached property trick comes in very handy for this.

The lines between DataModels and ViewModels can be blurry. DataModels are often shown in the UI with some DataTemplate, which isn’t really so different than the way we use ViewModels. However, the distinction usually makes sense in practice. I also want to point out that there’s often composition at many layers. ViewModels may compose other ViewModels and DataModels. And, DataModels may be composed of other DataModels.

The **ViewModel** is a model of the view. That means: You want to DataBind a property from your DataObject (model) to a property from your ViewObject (view) but you sometimes cannot bind directly to a CLR property of the model (because of converting or calculating). This is when ViewModel comes into play. It propagates the already calculated or converted value from your model, so you can bind this property directly to the view property.

The main thrust of the Model/View/ViewModel architecture seems to be that on top of the data (”the Model”), there’s another layer of non-visual components (”the ViewModel”) that map the concepts of the data more closely to the concepts of the view of the data (”the View”). It’s the ViewModel that the View binds to, not the Model directly.

The model

Using the **INotifyPropertyChanged** you can bubble changes up the stack. The reason that public methods should be on the UI thread is because the model could call long running or async stuff which would block the UI, though there are methods to let the UI thread handle property changes from a separate thread. See the doc on the Dispatcher object and the WPF threading model for more on this. Note in this context that you can let things happen in the background by means of the **BeginInvoke()** method of the Dispatcher and the paramter that specifies the priority. The SystemIdle in particular is interesting to be used when the Dispatcher is not busy.

The **DataModel** you can find in the download is mimiced from the Dan Crevier’s sample and can serve as an abstract base class for your own models.

System.Windows.Threading.Dispatcher (Message Pump) (Advance Topics)

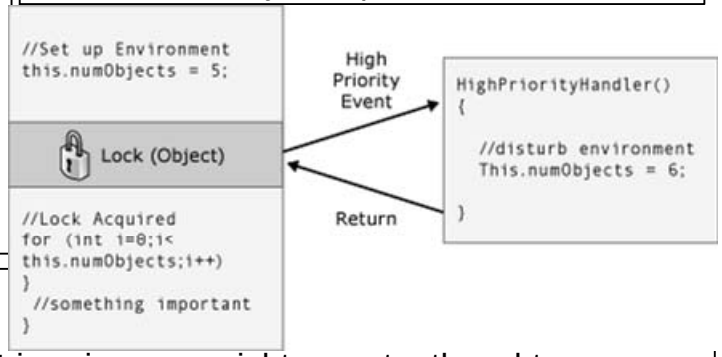
System.Windows.Threading.**Dispatcher** vs. System.Windows.Threading.**DispatcherSynchronizationContext**
Dipatcher is WPF specific, while DispatcherSynchronizationContext is not.

Nested Pumping

Sometimes it is not feasible to completely lock up the UI thread. Let's consider the Show method of the MessageBox class. Show doesn't return until the user clicks the OK button. It does, however, create a window that must have a message loop in order to be interactive. While we are waiting for the user to click OK, the original application window does not respond to user input. It does, however, continue to process paint messages. The original window redraws itself when covered and revealed.
Some thread must be in charge of the message box window. WPF could create a new thread just for the message box window, but this thread would be unable to paint the disabled elements in the original window (remember the earlier discussion of mutual exclusion). Instead, WPF uses a nested message processing system. The Dispatcher class includes a special method called PushFrame, which stores an application's current execution point then begins a new message loop. When the nested message loop finishes, execution resumes after the original PushFrame call.
In this case, PushFrame maintains the program context at the call to MessageBox.Show, and it starts a new message loop to repaint the background window and handle input to the message box window. When the user clicks OK and clears the pop-up window, the nested loop exits and control resumes after the call to Show.

Stale Routed Events

The routed event system in WPF notifies entire trees when events are raised. (If one handler takes a long time to finish, by the time the event gets to the 2nd handler on the tree, the event is really OLD.)



Reentrancy and Locking

http://msdn.microsoft.com/en-us/library/ms741870.aspx (Threading Model---WPF)

The locking mechanism of the common language runtime (CLR) doesn't behave exactly as one might imagine; one might expect a thread to cease operation completely when requesting a lock. In actuality, the thread continues to receive and process high-priority messages. This helps prevent deadlocks and make interfaces minimally responsive, but it introduces the possibility for subtle bugs. The vast majority of the time you don't need to know anything about this, but under rare circumstances (usually involving Win32 window messages or COM STA components) this can be worth knowing.
Most interfaces are not built with thread safety in mind because developers work under the assumption that a UI is never accessed by more than one thread. In this case, that single thread may make environmental changes at unexpected times, causing those ill effects that the DispatcherObject mutual exclusion mechanism is supposed to solve. Consider the following pseudocode:

Most of the time that's the right thing, but there are times in WPF where such unexpected reentrancy can really cause problems. So, at certain key times, WPF calls DisableProcessing, which changes the lock instruction for that thread to use the WPF reentrancy-free lock, instead of the usual CLR lock.
So why did the CLR team choose this behavior? It had to do with COM STA objects and the finalization thread. When an object is garbage collected, its Finalize method is run on the dedicated finalizer thread, not the UI thread. And therein lies the problem, because a COM STA object that was created on the UI thread can only be disposed on the UI thread. The CLR does the equivalent of a BeginInvoke (in this case using Win32's **SendMessage**). But if the UI thread is busy, the finalizer thread is stalled and the COM STA object can't be disposed, which creates a serious memory leak. So the CLR team made the tough call to make locks work the way they do.
The task for WPF is to avoid unexpected reentrancy without reintroducing the memory leak, which is why we don't block reentrancy everywhere.

"{StaticResource...}" vs <StaticResource ResourceKey=...>

They are just two forms of equivalent of referring to static resources

XAML Attribute Usage (MarkupExtension)

```
<object property="{StaticResource key}" .../>
```

Good for assigning a resource to a non-Collection property using **Property Attribute** syntax.

Style="{StaticResource myResourceName}"

XAML Object Element Usage

```
<object>
  <object.property>
<StaticResource ResourceKey="key" .../>
  </object.property>
</object>
```

Good for assigning resources to a **Collection** property using **Property Element** syntax.

```
<EventTrigger.Actions> // <<< Collection Property
    <StaticResource ResourceKey="myActionName"/>
</EventTrigger.Actions>
```

StaticResource let's one set a property of an element once. If the Desktop Color is changed while the element's application is running, the element keeps its original color:

```
<Button>
  <Button.Background>
    <SolidColorBrush Color="{StaticResource {x:Static SystemColors.DesktopColorKey}}" />
  </Button.Background>
  Hello
</Button>
```

DynamicResource, if the element's color is set using a DynamicResource, it changes when the Desktop Color changes:

```
<Button>
  <Button.Background>
    <SolidColorBrush Color="{DynamicResource {x:Static SystemColors.DesktopColorKey}}" />
  </Button.Background>
  Hello
</Button>
```

Why is that? The answer comes from the way these two Resource finders work:

1. StaticResource – Finds the resource given in by the ResourceDictionary key, and keeps the resource value;
2. DynamicResource – Finds the resource in the ResourceDictionary and keeps the key.

Since DynamicResource keeps the resource Key instead of the resource value, every event change fired from the resource lets the DynamicResource know that its value has changed. This is why the DynamicResource reacts to the resource changes, while the StaticResource can't know the resource has changed, since it only keeps the resource's final value.

http://wpfxaml.spaces.live.com/blog/cns!97DD5FD32788695B!142.entry

StaticResource vs DynamicResource

Static- keeps the value while Dynamic- keeps the key
Static- happens at XAML load-time while Dynamic- at runtime

StaticResource - Provides a value for a XAML property by substituting the value of an already defined resource	**DynamicResource** - Provides a value for a XAML property by deferring that value to be a runtime reference to a resource. A dynamic resource reference forces a new lookup each time that such a resource is accessed.
A StaticResource will be resolved and assigned to the property during the loading of the XAML which occurs before the application is actually run. It will only be assigned once and any changes to resource dictionary ignored.	A DynamicResource assigns an Expression object to the property during loading but does not actually lookup the resource until runtime when the Expression object is asked for the value. This defers looking up the resource until it is needed at runtime. A good example would be a forward reference to a resource defined later on in the XAML. Another example is a resource that will not even exist until runtime. It will update the target if the source resource dictionary is changed.

http://stackoverflow.com/questions/200839/whats-the-difference-between-staticresource-and-dynamicresource-in-wpf

"{Binding ...}" vs <Binding>

They are just two equivalent forms

XAML Attribute Usage (MarkupExtension)

```
<object property="{Binding}" .../>
-or-
<object property="{Binding
  bindingPropertyName1=value,
  bindingPropertyName2=value,
  bindingPropertyNameN=value}" ...
/>
```

XAML Object Element Usage

```
<object>
  <object.property>
    <Binding/>
  </object.property>
</object>
-or-
<object>
  <object.property>
    <Binding
      bindingPropertyName1="value"
      bindingPropertyName2="value"
      bindingPropertyNameN="value"
    />
  </object.property>
</object>
```

http://msdn.microsoft.com/en-us/library/system.windows.data.binding.aspx

Event vs Command vs Trigger

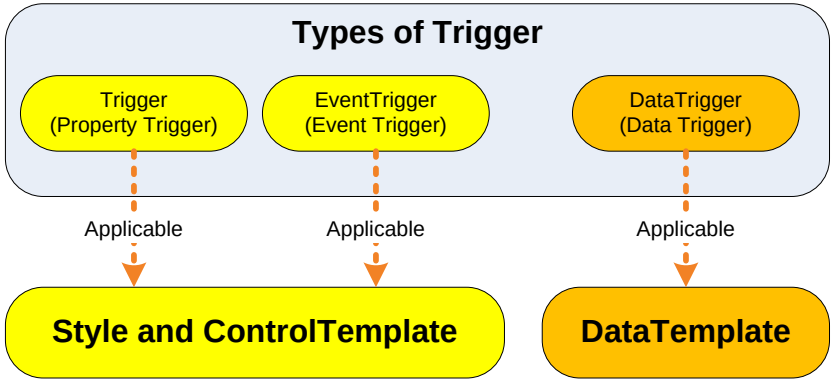
At the spectrum of coupling actions, Event and Command are at the opposite sides; event being the most tightly coupled, while Command is the least tightly coupled.

Event – include Low-Level Physical events (MouseDown, MouseUp, etc) and High-Level, Semantic events (Click, DoubleClick)

It is better to work with Semantic events than Physical events, if possible.

Command – ICommand-Based or RoutedCommand based (really Routed Event-based) to enable decoupling command logic from Event and from Command object. (Refer to Commanding worksheet)

Triggers only applied in Templates and Style:



Essential Windows Presentation Foundation – Chris Anderson (Ch 7 Actions)

WPF Perf: RenderCapability.Tier & DesiredFrameRate

In this post, I'm going to talk about two key API's for performance in WPF. These are RenderCapability.Tier and Storyboard.DesiredFrameRate. In this post, I'm going to show:

How to leverage RenderCapability.Tier to scale your app up or down.
How to use RenderCapability.Tier in markup.
How to apply DesiredFrameRate to reduce CPU consumption.

RenderCapability.Tier

For the machine on which it's run, RenderCapability.Tier signals the machine's hardware capabilities. Tier=0 means software rendering; Tier=2 is hardware rendering (for those features that can be rendered in hardware); Tier=1 is a middle ground (some things in hardware and some in software.) You can probably see how this might be useful to scale up or down the richness of an application depending on the hardware.

(Note: RenderCapability.Tier is an integer value using the high word to indicate the tier. You'll need to shift by 16 bits to get the corresponding tier values. Adam Smith posts about why this was chosen.)

Storyboard.DesiredFrameRate

The second API I mentioned was Storyboard.DesiredFrameRate (DFR.) DFR allows you to specify, manually, what the frame rate "should" be (the animation system attempts to get as close as possible to the DFR value but there are no guarantees.) By default, WPF attempts 60 fps. The up side to this is that animations look better; the down side is that you may not need 60 fps but you may be spending the extra CPU cycles.

Putting Them Together

You can use RenderCapability.Tier in markup with a small helper class. The trick is to wrap up the property in a DependencyProperty. From there, the property can be used to set DFR. Below, you'll find both the code for the RenderCapability.Tier, how to use it in markup and how to set DesiredFrameRate. I picked different DFR values for the different Tiers. The resulting UI isn't that exciting (a spinning Button) but it demonstrates the concept.

```
using System;
using System.Windows;
using System.Windows.Media;
namespace PerformanceUtilities{
    public static class RenderCapabilityWrapper {
        public static readonly DependencyProperty TierProperty = DependencyProperty.RegisterAttached("Tier",
                                                                  typeof(int),
                                                                  typeof(PerformanceUtilities.RenderCapabilityWrapper),
                                                                  new PropertyMetadata(RenderCapability.Tier >> 16));

        public static int GetTier(DependencyObject depObj){
            return (int)TierProperty.DefaultMetadata.DefaultValue;
        }
    }
}
```

```
<Border xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation" xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
xmlns:Perf="clr-namespace:PerformanceUtilities;assembly=PerfTier" Background="silver">
    <Border.Resources>
        <Style x:Key="{x:Type Button}" TargetType="{x:Type Button}">
            <Style.Triggers>
                <Trigger Property="Perf:RenderCapabilityWrapper.Tier" Value="2"> <Setter Property="Tag" Value="60"/> </Trigger>
                <Trigger Property="Perf:RenderCapabilityWrapper.Tier" Value="1"> <Setter Property="Tag" Value="30"/> </Trigger>
                <Trigger Property="Perf:RenderCapabilityWrapper.Tier" Value="0"> <Setter Property="Tag" Value="15"/> </Trigger>
            </Style.Triggers>
        </Style>
    </Border.Resources>

    <Button Width="80" Height="80" Name="MyButton"
        Content="{Binding RelativeSource={RelativeSource Self}, Path=Tag}">
        <Button.Triggers>
            <EventTrigger RoutedEvent="FrameworkElement.Loaded">
                <BeginStoryboard>
                    <Storyboard RepeatBehavior="Forever"
                        Storyboard.DesiredFrameRate="{Binding ElementName=MyButton,Path=Tag}">
                        <DoubleAnimation Storyboard.TargetName="MyAnimatedTransform"
                        Storyboard.TargetProperty="(RotateTransform.Angle)"
                            From="0.0" To="360" Duration="0:0:10" />
                    </Storyboard>
                </BeginStoryboard>
            </EventTrigger>
        </Button.Triggers>
        <Button.RenderTransform> <RotateTransform CenterX="40" CenterY="40" x:Name="MyAnimatedTransform"/> </
        Button.RenderTransform>
    </Button>
</Border>
```

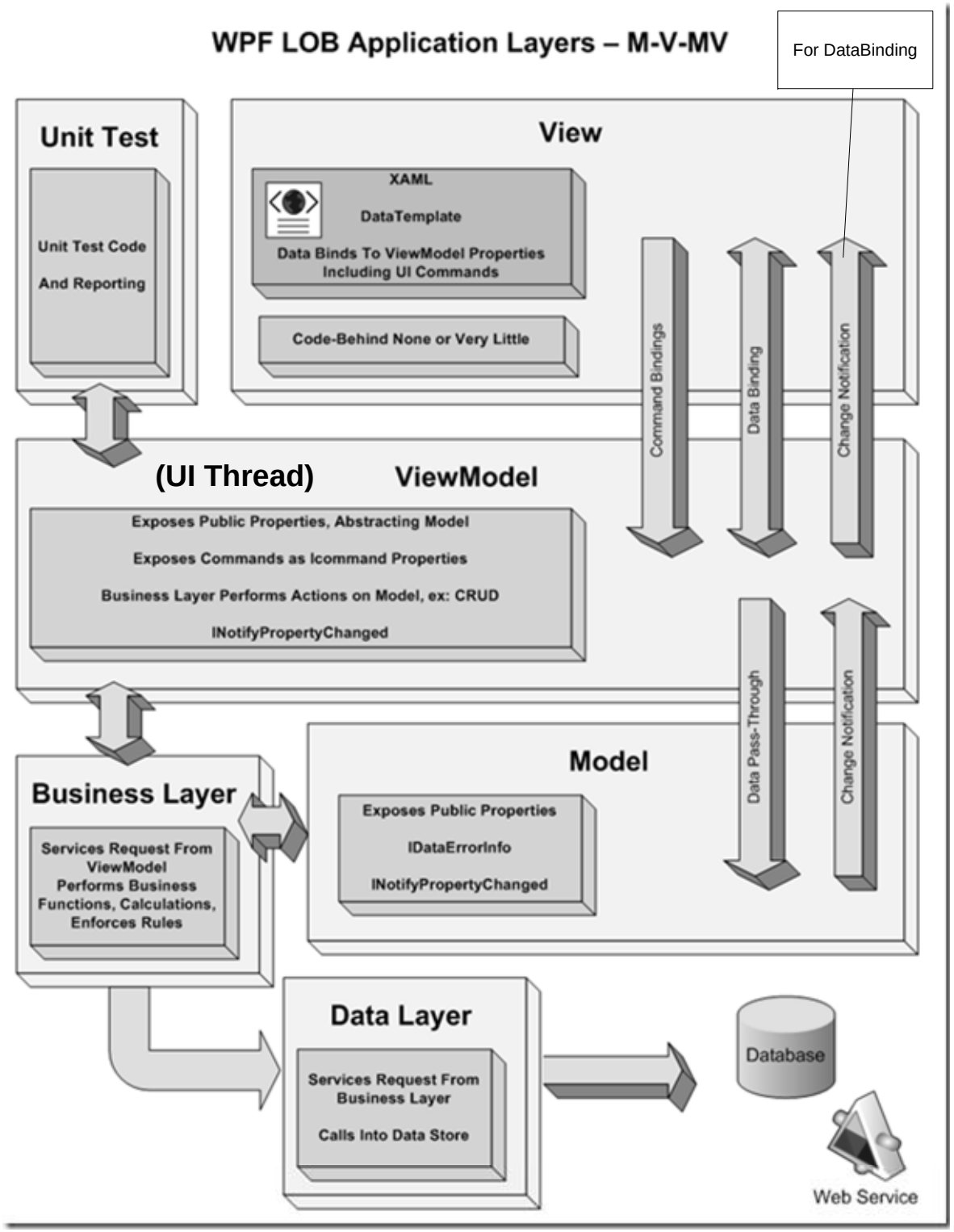
WPF Perf: RenderCapability.Tier & DesiredFrameRate

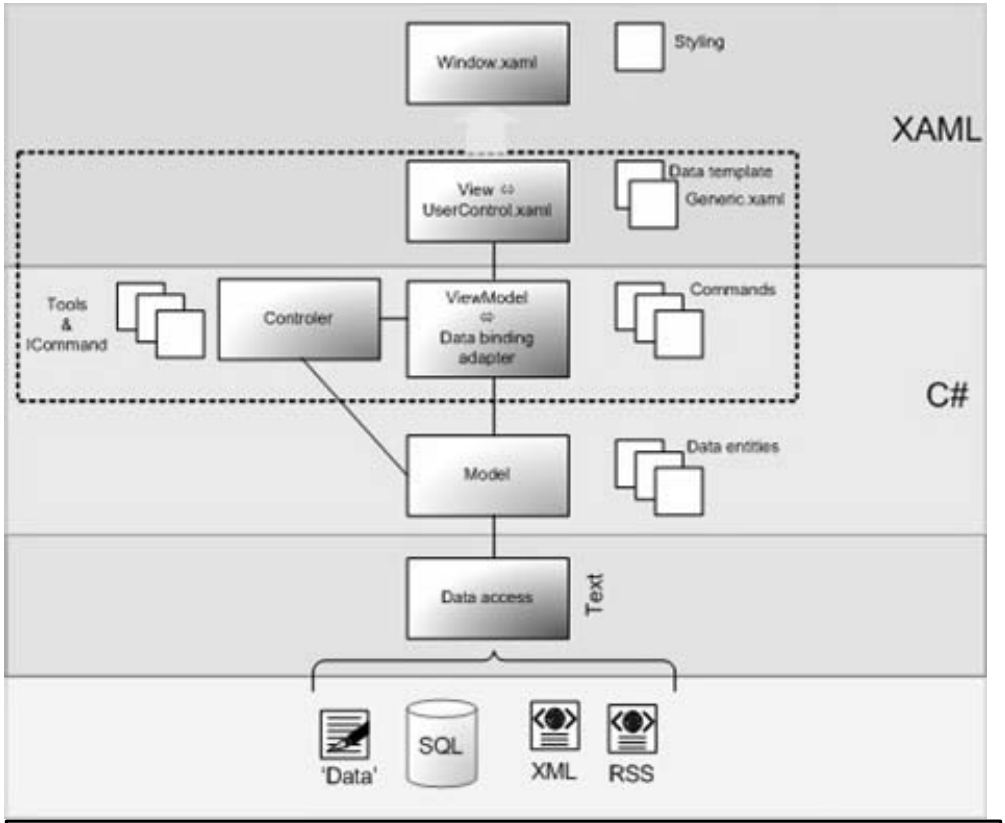
http://blogs.msdn.com/henryh/archive/2006/08/23/719568.aspx

User Control vs Custom Control

User Control base on UserControl
1) usercontrol.xaml
2) usercontrol.xaml.cs

Custom Control
1) customControl.cs
2)Themes/generic.xaml (default controlTemplate)





WorkFlow of MVVM:

1) Define a View in Xaml.
2) In Xaml or in Code, Setup View.DataContext to use corresponding ViewModel instance.
3) In Xaml, DataBind UI Element's Properties to corresponding "State" Properties on the ViewModel instance (Mode could be OneWay or TwoWay, depending on the functionality)
4) In Xaml, Databind UI Elements "Command" to ViewModel's Command Property. At the same time, specify the "CommandParameter" via DataBind or ???
5) Code ViewModel.cs, according to MVVM pattern

* Repeat the above for each View

The Essence of Model.View.ViewModel (MVVM)

http://www.codeproject.com/KB/architecture/MVC_MVP_MVVM_design.aspx

Step 1 **View.xaml (View Definition)**

```
<CONTAINER …        // <<< Window, Page, UserControl
  x:Class=NAMESPACE.View
  xmlns=…
  xmlns:x=…
  xmlns:vm=”clr-namespace:NAMESPACE;”
  ...
  >
  <CONTAINER.DataContext>
1   <vm:ViewModel/>  // << can ONLY call ViewModel() default ctor
  </CONTAINER.DataContext>

  <PANEL>  // <<< Grid, StackPanel, etc.
    <TextBlock Text=”{Binding State}” />
    <Button Command=”{Binding XyzCommand}”>Update</Button>
    ...
  </PANDEL>
</CONTAINER>
```

Instantiate a ViewModel via default ctror

DataBind to ViewModel’s State

DataBind to ViewModel’s XyzCommand

DataBinding!

Step 2 **View.xaml.cs (code-behind)**

```
public partial class View : CONTAINER {

  // Setting up DataContext in Code allows
  // calling non-default ViewModel ctor!
2 DataContext = new ViewModel(param);

  InitializeComponent();
}
```

Instantiate a ViewModel via non-default ctor (not shown)

View.g.cs (generated)

```
public partial class View : CONTAINER {

  public void InitializeComponent() {
    // ...
  }
  // ...
}
```

Calls

ViewModel.cs **Step 3**

```
public class ViewModel : INotifyPropertyChanged {  // INotifyPropertyChanged is for DataBinding

  private TYPE state;
  private ICommand xyzCommand;
  private Model model = new Model(); // Model code is not shown

  public TYPE State {
    get { return state;}
    set { state = value; OnPropertyChanged(“State”); }
  }

  public ICommand XyzCommand {
    get {
      if (xyzCommand == null)
        xyzCommand = new ViewModelCommand((p)=>DoCommand(p), (p)=>CanDoCommand(p));
      return xyzCommand;
    }
    private set { xyzCommand=value;}
  }

  private void DoCommand(object p) {…}
  private bool CanDoCommand(object p) {...}

  public event PropertyChangedEventHandler PropertyChanged;
  protected void OnPropertyChanged(string propertyName) {
    if (PropertyChanged != null)
      PropertyChanged(this, new ProertyChangedEventArgs(propertyName));
  }
}

public sealed class ViewModelCommand : ICommand {
  readonly Action<object> m_execute;
  readonly Predicate<object> m_canExecute;

  public ViewModelCommand(Action<object> execute) : this(execute, null) {}
  public ViewModelCommand(Action<object> execute, Predicate<object> canExecute) {
    if (execute == null)
      throw new ArgumentNullExeception(“execute”);
    m_execute = execut; m_canExecute = canExecute;
  }
  [DebuggerStepThrough]
  public bool CanExecute(object param) {
    return m_canExecute == null ? true : m_canExecute(param);
  }
  public event EventHandler CanExecutedChanged {  // Observers to add their delegates
    add {CommandManager.RequerySuggested += value;}
    remove {CommandManager.RequerySuggested -= value;}
  }
  public void Execute{object param) { m_execute(param);
}
```

ViewModel’s State and Command for being bound to View

Delegate

Delegate

Instantiate a Command

ref to

ref to

Change

(p as Type)
Real-Work of ViewModel is Done HERE!

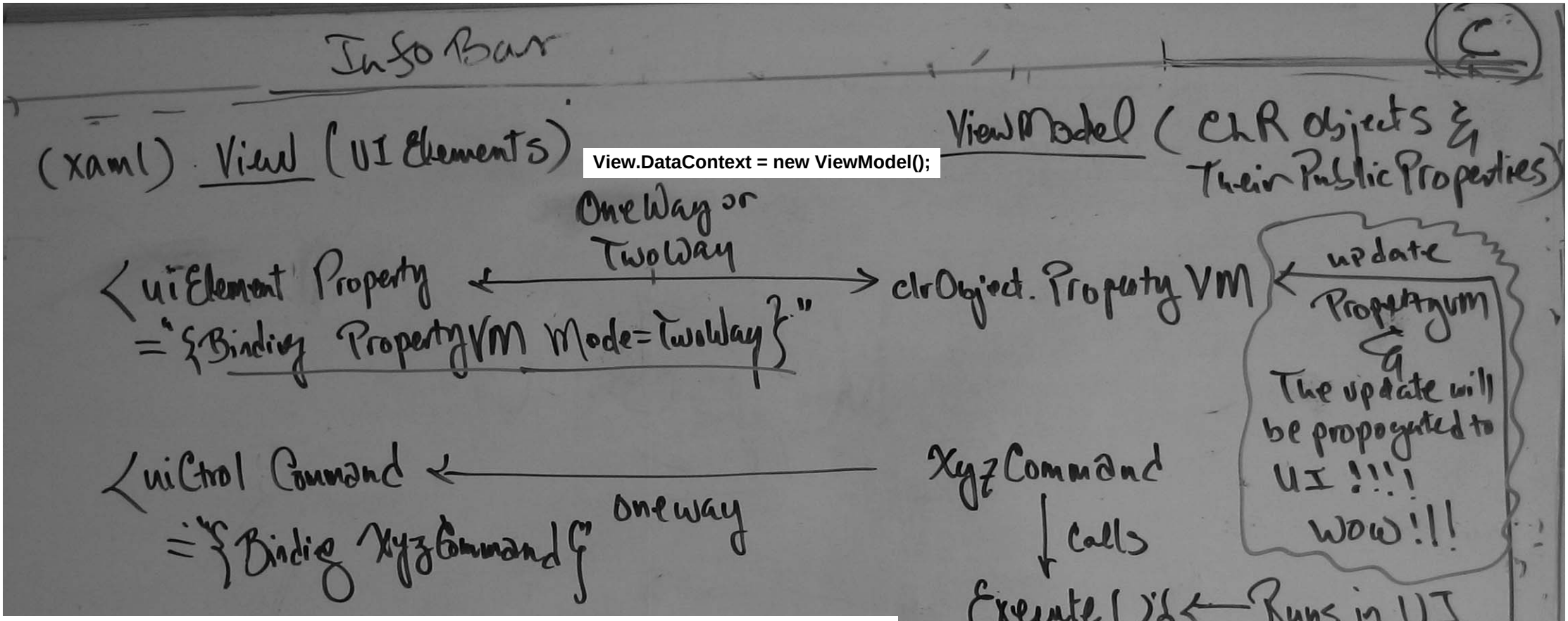
One Set per Feature Set
Programmer Work

ViewModelCommand Instance

Implements

Boilerplate Code!

Note: **Two ways to set View.DataContext to ViewModel instance 1) in Xaml; 2) in Code (which can use ViewModel’s non-default ctor)**



The Beauty of View + ViewModel + Threading (Wow!!!)

No more Hassle to deal with UI Thread in Background Threads

Wow!!!!
In the Thread, we can free to update the CLR object's PropertyVM! No Worry about UI Thread Affinity Issue!

While experimenting with WPF data binding mechanism today, I've noticed a feature which could ease the pain of following the core principle of GUI multi-threaded programming – "Thou shalt not interact with a control's properties from a thread other than the one that created the control."

In my little WPF databinding experiment I created a data class implementing INotifyPropertyChanged interface, and a simple WPF window with single textbox on it. My WPF window has an instance of my data class and the textbox's Text property is bound to 'AskPrice' property of my data class. To be more concrete, here is the source code:

Data Class:

```
public class Data : INotifyPropertyChanged {
    private double askPrice;
    public double AskPrice {
        get { return askPrice; }
        set {
            askPrice = value;
            PropertyChangedEventHandler temp = PropertyChanged;
            if (temp != null ) { temp(this, new PropertyChangedEventArgs("AskPrice"));}
        }
    }
    public event PropertyChangedEventHandler PropertyChanged;
}
```

WPF window XAML:

```
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
Title="Window1? Height="99? Width="140? Loaded="Grid_Loaded">
```

WPF Window Code Behind:

```
public partial class Window1 : Window {
    private Data data;
    private Thread dataSupplier; //Background Thread updating the data
    public Window1() {
        InitializeComponent();
        data = new Data(); // << Instantiated in UI-Thread!
        ask.SetBinding(TextBox.TextProperty, new Binding("AskPrice"));
        dataSupplier = new Thread(new ThreadStart(this.PumpData));
    }
    public void PumpData() {
        while (true) {
            Random r = new Random();
            data.AskPrice = r.NextDouble();
            Thread.Sleep(1000);
        }
    }
    private void Grid_Loaded(object sender, EventArgs e) {
        this.Grid1.DataContext = data;
        dataSupplier.Start();
    }
}
```

Path

Action<Double> SetAskPrice = (r)=> {data.AskPrice = r.NextDouble(); };
_disp.BeginInvoke(SetAskPrice, DispatcherPriority.Send, null); // **Ugly!!!**

Used To Be Like This

Run in a Thread context !!

WPF, Data Binding & Multithreading June 24th, 2007

http://blog.lab49.com/archives/1166

MVVM (DataBinding) + Threading--- Why It Works? Explained

If you're still reading so far with your .NET 2.0/1.1/1.0 glasses, you might have noticed that 'PumpData' function is violating the cardinal rule of Windows GUI programming by setting the Text property (via DataBinding) on a non-GUI thread. If you port the above code to .NET 2.0 you will get the good old "InvalidOperationException was unhandled" error message. However, in .NET 3.5 this code works without any errors and you will see that your ask price is ticking nicely. It seems that WPF framework is marshalling the background thread update to the GUI thread behind scenes, but how?

The explanation starts to appear when you put a break point in "PumpData" function and debug the application. If you check the value of 'ask.GetBindingExpression(TextBox.TextBoxProperty)' in your watch window and expand the non-public members of it, you will see that BindingExpression has an instance of mysterious "ClrBindingWorker" class and that instance contains the 'Dispatcher' which is responsible from marshalling background thread calls onto GUI thread.

Unfortunately, when you check the MSDN or Google for "ClrBindingWorker" class, you cannot get any helpful information. As a last result, I tried Reflector to see what this undocumented class does behind the scenes. ClrBindingWorker is derived from BindingWorker class and a quick Reflector investigation reveals that it is mainly responsible from attaching binding to data item and performing the data transfer. Among its more than 20 member functions two of them seem interesting to explain behind data marshalling, "private static object OnCompleteGetValueCallback(AsyncDataRequest adr)" and "private static object OnCompleteSetValueCallback(AsyncDataRequest adr)". When you disassemble these:

```
private static object OnCompleteSetValueCallback(AsyncDataRequest adr) {
    AsyncSetValueRequest arg = (AsyncSetValueRequest) adr;
    ClrBindingWorker worker = (ClrBindingWorker) arg.Args[0];
    worker.Dispatcher.BeginInvoke(DispatcherPriority.DataBind,
                                  CompleteSetValueLocalCallback, arg);
    return null;
}

private static object OnCompleteGetValueCallback(AsyncDataRequest adr) {
    AsyncGetValueRequest arg = (AsyncGetValueRequest) adr;
    ClrBindingWorker worker = (ClrBindingWorker) arg.Args[0];
    worker.Dispatcher.BeginInvoke(DispatcherPriority.DataBind,
                                  CompleteGetValueLocalCallback, arg);
    return null;
}
```

The Dispatcher instance is used to marshal the execution of background thread call onto the UI thread!

Attached Behavior

View.xaml (View Definition)

```
<Window x:Class="WpfApplication1.Window1"
    xmlns="..."
    xmlns:x="…"
    xmlns:local="clr-namespace:WpfApplication1"
    Title="Window1" Height="300" Width="300">
    <Grid>
        <Button Content="Hello"
            local:ClickBehavior.RightClick="{Binding Foo}"/>
    </Grid>
</Window>
```

ViewModel.cs (ViewModel)

```
public class ViewModel {
  public ICommand Foo {
    get { return new DelegateCommand(this.DoSomeAction);}
  }

  private void DoSomeAction() {
    MessageBox.Show("Command Triggered");
  }
}
```

View.xaml.cs (Coe Behind)

```
public partial class Window1 : Window {
  public Window1() {
    InitializeComponent();
    this.DataContext = new ViewModel();
  }
}
```

ClickBehavior.cs (ViewModel)

Attached Property

Attached Property Pattern

```
public static class ClickBehavior {
  public static DependencyProperty RightClickCommandProperty =
    DependencyProperty.RegisterAttached(
        "RightClick", typeof(ICommand), typeof(ClickBehavior),
      new FrameworkPropertyMetadata(null,
          new PropertyChangedCallback(ClickBehavior.RightClickChanged)));

  public static void SetRightClick(DependencyObject target, ICommand value) {
    target.SetValue(ClickBehavior.RightClickCommandProperty, value);
  }

  public static ICommand GetRightClick(DependencyObject target) {
    return (ICommand)target.GetValue(RightClickCommandProperty);
  }
```

Property Changed Callback

```
  private static void RightClickChanged(
      DependencyObject target, DependencyPropertyChangedEventArgs e) {
    UIElement element = target as UIElement;

    if (element != null) {
      if ((e.NewValue != null) && (e.OldValue == null)) {
        // We're putting in a new command if there wasn't one hook already
        element.MouseRightButtonUp += element_MouseRightButtonUp;
      } else if ((e.NewValue == null) && (e.OldValue != null)) {
        // We're clearing the command if it wasn't already null unhook
        element.MouseRightButtonUp -= element_MouseRightButtonUp;
      }
    }
  }
```

PropertyChangeCallback to add or remove an MouseButtonEventHandler to the UI's MouseRightButtonUp event

EventHandler Calling Cmd!

Inside MouseRightButtonUp eventhandler, retrieve the Command and execute it!

```
  static void element_MouseRightButtonUp(object sender, MouseButtonEventArgs e){
    UIElement element = (UIElement)sender;
    ICommand command = (ICommand)element.GetValue(
                                        ClickBehavior.RightClickCommandProperty);
    command.Execute(null);
  }
}
```

ViewModel.cs (ViewModel)

```
public class DelegateCommand : ICommand {

  public delegate void SimpleEventHandler();
  private SimpleEventHandler handler;
  private bool isEnabled = true;
  public event EventHandler CanExecuteChanged;
  public DelegateCommand(SimpleEventHandler handler) {
    this.handler = handler;
  }

  private void OnCanExecuteChanged() {
    if (this.CanExecuteChanged != null) {
      this.CanExecuteChanged(this, EventArgs.Empty);
    }
  }

  bool ICommand.CanExecute(object arg){
    return this.IsEnabled;
  }

  void ICommand.Execute(object arg) {this.handler();}

  public bool IsEnabled {
    get { return this.isEnabled; }
    set {
      this.isEnabled = value;
      this.OnCanExecuteChanged();
    }
  }
}
```

Warning: Way to Complicated. Don't Use it! Not Worth the Complexity

http://blogs.msdn.com/johngossman/archive/2008/05/16/attachedbehavior-pattern-sample.aspx

Theory of Attached Behavior (via Attached Property Pattern)

ViewModelCommand (aka RelayCommand and DelegateCommand) is used to be DataBound to controls' Command properties. ViewModelCommand instances that contains command handlers which lives in the ViewModel.

Q: Why need Attached Behavior when we already have the ViewModelCommand?
A: ViewModelCommand can only be DataBound to Controls that implement ICommandSource. In other words, if a control does not implement ICommandSource, then we can not do DataBinding it to a ViewModelCommand. In addition, even if a Control implements ICommandSource not all its events are related to the Command. For example, Button implements ICommandSource, but only the "Click" event trigger the Command execution, whereas DoubleClick and RightClick events are not tied to any Command execution. Attached Behavior pattern is used to allow us to DataBind any control's any event to a Command. For example, we want to DataBind Button's DoubleClick event to a SaveCommand that gets executed on ViewModel layer. However, the DoubleClick event CAN NOT DIRECTLY take an ICommand object as its value. That leads to the next Question.

Q: How a Control's Event, Attached Property and a ViewModelCommand are tied together?
A: Basically, by doing the following, we are hooking up a Control's event with its event handler (on ViewModel) which calls the Command handler (which also on ViewModel) which get called when the Event Raised on the Control.

1) Define a static class ViewModelBehavior, which has an Attached Property (name after the target EventName on the Control, eg: DoubleClick)

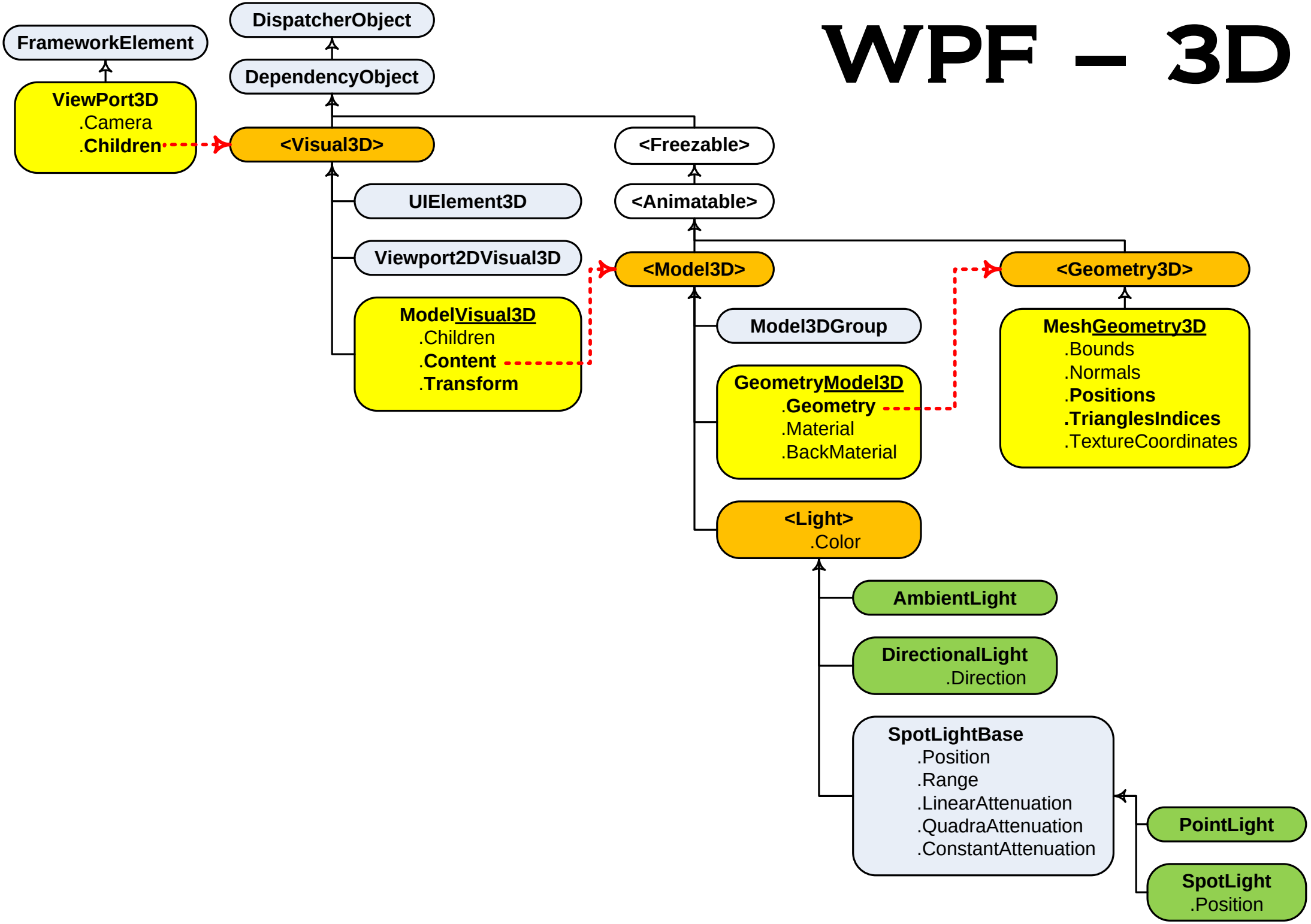
2) In the Attached Property, instantiate a PropertyChangedCallback delegate that points to a static function in the ViewModelBehavior class; the static function will setup the Target Control's EventHandler for the given Event (either add or remove).
3) In that eventhandler, calls the retrieve the Command stored inside the target UI element, and execute it.

Basically, it is a fancy way to setup an Event Property of a Control to ViewModel's EventHandler!

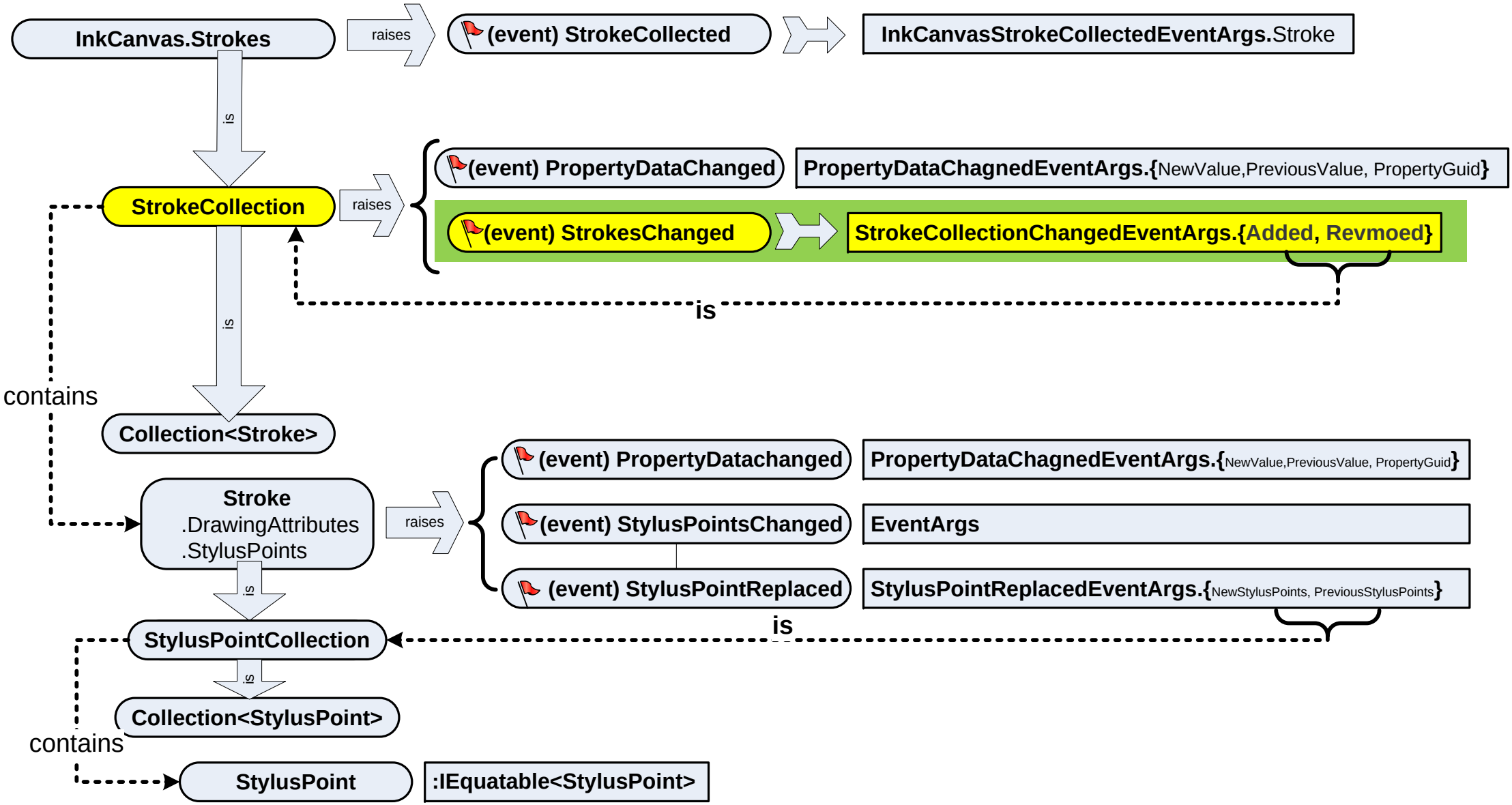
Q: Wouldn't it easier to just assign the ViewModel's eventHandler directly to the UI element's Event? What is the benefit for the fuss? Plus, for different events (of different delegate types), we have to create different ViewModelBehavior! That means more code to maintain---even though the codes are pretty much boilerplate.

Conclusion: Don't use Attached Behavior, use ViewModel directly as the listener to hookup a Control's event!

WPF – 3D


FrameworkElement
DispatcherObject
DependencyObject
ViewPort3D
.Camera
.Children
<Visual3D>
<Freezable>
<Animatable>
UIElement3D
Viewport2DVisual3D
ModelVisual3D
.Children
.Content
.Transform
<Model3D>
<Geometry3D>
Model3DGroup
GeometryModel3D
.Geometry
.Material
.BackMaterial
MeshGeometry3D
.Bounds
.Normals
.Positions
.TrianglesIndices
.TextureCoordinates
<Light>
.Color
AmbientLight
DirectionalLight
.Direction
SpotLightBase
.Position
.Range
.LinearAttenuation
.QuadraAttenuation
.ConstantAttenuation
PointLight
SpotLight
.Position

InkCanvas & StrokeCollection



Control Template