WPF Framework Overview

WPF From 10,000 Feet

This article provides a high level overview of the WPF framework. The topics introduced in this document will be expanded upon in more detailed documents.

1. WPF executables – Windows and UI Threads
2. XAML
3. Resources
4. Logical and Visual Trees (Templating)
5. Styles and Triggers
6. Dependency Properties
7. DataBinding and the DataContext
8. Controls s

Dependencies and Building

The WPF framework is spread over three Dll's

- PresentationCore.dll
- PresentationFramework.dll
- WindowsBase.dll

For most .NET project we also need to include `System.xam.dll` A simple bare bones WPF MainClass.cs is as follows.

Listing 1Basic MainClass

```
using System;
using System.Windows;
using System.Windows.Threading;

public class MainClass
{
    [STAThread]
    private static void Main(String[] args)
    {
        Application application = new Application();
        bool b = application == Application.Current;
        Window w = new Window();
        application.Run(w);
    }
}
```

The Window

The first point is that a WPF application needs to contain at least one window. It is after all a user interface application. The window in a WPF application is really just a Win32 window. The OS doesn't know the difference between a window with Win32 content and a window with WPF content.

We can create any number of windows. By default, as each window is closed the other windows remain running. If we want to create a *modeless dialog* which closes when its parent closes, we need to set the child windows `Owner` property to be the parent window.

Threading Model

A Win32 window has to run inside a thread whose COM apartment state is `ApartmentState.STA` We achieve this by marking the entry method Main with the following attribute.

```
[System.STAThread]
public static void Main(string[] args)
```

To make our new STAThread a UI Thread it needs to have an associated message loop. The message loop is setup by calling. The message loop sits inside a loop, pulling messages from a message queue and dispatching them. Any window's input messages such as keystrokes and mouse presses are pushed onto the message queue by the operating system. It is then the message loop's job to pull these from the queue, process them and dispatch them such that handlers can respond to them. The following line of code makes the thread it is executing on a UI thread,

```
System.Windows.Threading.Dispatcher.Run();
```

Any WPF object whose type extends `DispatcherObject` has what is known as thread affinity. It must be created on a UI thread and once created it has affinity with that UI thread. It can only be safely accessed from the user interface thread that created it.

If any thread other than the user interface thread wants to interact with elements created by the user interface thread in a thread safe fashion, they must do so via that elements Dispatcher. The DispatcherObject type defines a Dispatcher which can be used for this purpose. Almost all WPF types extend the type `DispatcherObject` and as such have access to the dispatcher associated with the thread which created them.

The Application

Although not mandatory, most WPF executables will make use of an instance of the type Application .

> **ONE APPLICATION PER APP DOMAIN**

WPF ensures there is only Application per app domain. If we try to instantiate a second Application

XAML

It is possible to build WPF user interfaces using just procedural code. In practice, WPF user interfaces tend to be built using XAML. The details of XAML are beyond this basic introduction however the following listing gives a flavout of a very basic user interface consisting of a window and a button.

Listing 2Simple XAML

```
<Window x:Class="MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="MainWindow">
    <Button>Kenny</Button>
</Window>
```

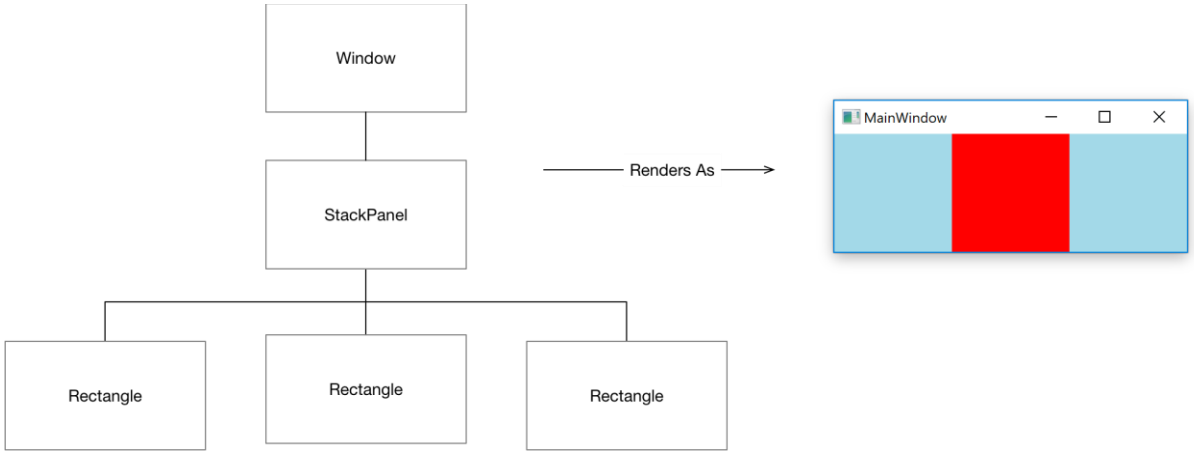
For more information on XAML see Xaml - Detail

The logical tree

A WPF Application XAML is used to build up a tree of elements know as the logical tree. The logical tree is fundamental to understanding many of the features of WPF. Consider the following piece of XAML

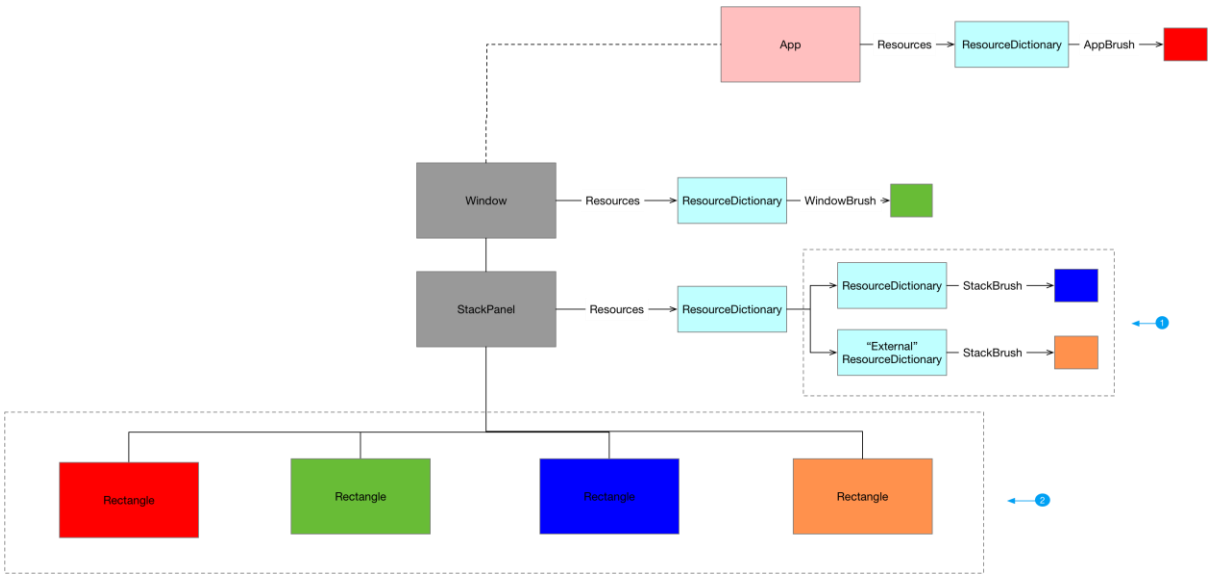
```
<Window x:Class="LogicalTree.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
        xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
        mc:Ignorable="d"
        Title="MainWindow" SizeToContent="WidthAndHeight">
    <StackPanel Orientation="Horizontal">
        <Rectangle Width="100" Height="100" Fill="LightBlue"></Rectangle>
        <Rectangle Width="100" Height="100" Fill="White"></Rectangle>
        <Rectangle Width="100" Height="100" Fill="LightBlue"></Rectangle>
    </StackPanel>
</Window>
```

This piece of XAML corresponds to the following logical tree and rendering



Resource Mechanism

Resource Dictionaries provide a standard mechanism for packaging and accessing resources in WPF. We extend the simpe piece of XML from the previous section to show how resources work.



1. Resource dictionaries can be merged
2. Framework Elements can access resources from the resources collections of parents in the logical tree (and resources defined in the application itself)

   Resources can be accessed as either static or dynamic resources
   Below we share the source code for this example

Listing 3ExternalResourceDict.xaml

```
<ResourceDictionary xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
                    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
    <SolidColorBrush x:Key="ExternalResourceBrush" Color="Orange"/>
</ResourceDictionary>
```

Listing 4App.xaml

```
<Application x:Class="Resources.App"
             xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
             xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
             StartupUri="MainWindow.xaml">
    <Application.Resources>
        <SolidColorBrush x:Key="AppBrush" Color="Red"></SolidColorBrush>
    </Application.Resources>
</Application>
```

Listing 5MainWindow.xaml

```
<Window x:Class="Resources.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="MainWindow" SizeToContent="WidthAndHeight">
    <Window.Resources>
        <SolidColorBrush x:Key="WindowBrush" Color="Green"></SolidColorBrush>
    </Window.Resources>

    <StackPanel Orientation="Horizontal">
        <StackPanel.Resources>
            <!-- 1. Resources can be merged -->
            <ResourceDictionary>
                <ResourceDictionary.MergedDictionaries>
                    <ResourceDictionary>
                        <SolidColorBrush x:Key="StackBrush" Color="Blue"/>
                    </ResourceDictionary>
                    <ResourceDictionary Source="ExternalResourceDict.xaml"/>
                </ResourceDictionary.MergedDictionaries>
            </ResourceDictionary>
        </StackPanel.Resources>

        <!-- FrameworkElements can access resources from the resource collections
        of any ancestor in the logical tree and the App resources itself -->
        <Rectangle Width="100" Height="100" Fill="{StaticResource AppBrush}"></Rectangle>
        <Rectangle Width="100" Height="100" Fill="{StaticResource WindowBrush}"></Rectangle>
        <Rectangle Width="100" Height="100" Fill="{StaticResource StackBrush}"></Rectangle>
        <Rectangle Width="100" Height="100" Fill="{StaticResource ExternalResourceBrush}"></Rectangle>
    </StackPanel>
</Window>
```

For more information on the resource mechanism see [Resource - Detail]

Mapping Logical tree to Visual Tree

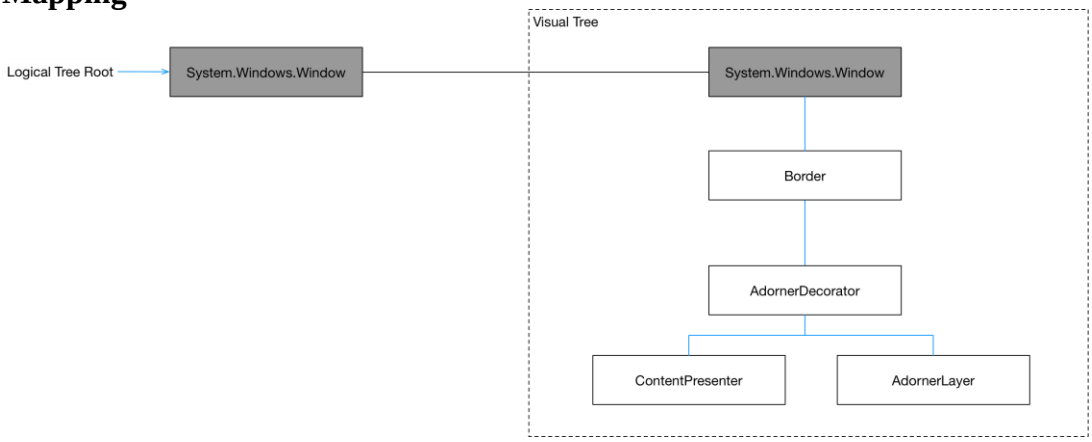
The visual tree is an expansion of the logical tree into core visual components. A very simple UI may consist of just a single node logical tree holding the Window itself. This window is however made up of multiple visual objects that render it. Consider a basic window as follows.

SINGLE LOGICAL TO MULTI-NODE VISUAL

Source code

```
<Window x:Class="LogicalToVisualOne.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="MainWindow" Height="200" Width="200">
</Window>
```

Mapping



Controls

The killer feature of WPF controls is that the procedural behaviour code is completely separated from its visual tree. Every Control has a `Template` dependency property of type `ControlTemplate`. A control template enables the consumer of a control to replace its visual tree with a completely new and arbitrarily complex tree of visuals while keeping the core behaviour of the control intact. Whenever a Control is instantiated its ControlTemplate is used to generate a tree of visuals that will be used in rendering. The template acts as a blueprint that tells WPF how to create the visual elements needed to render the Control. The content of a ControlTemplate is of type `VisualTree`.

All FrameworkElements in a control template have a relationship back to the control being templated. This relationship is know as the templated parent and is represented by the property

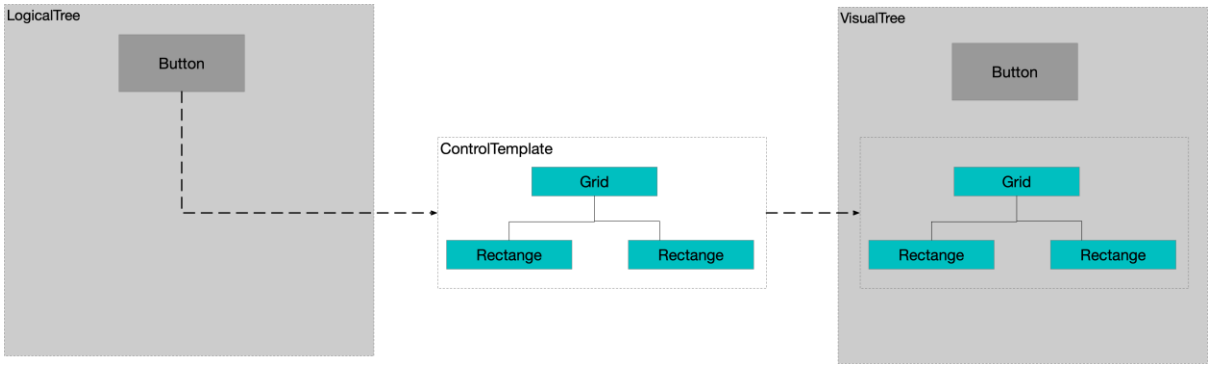
CREATING THE CONTROLTEMPLATE

Listing 6Creating a ControlTemplate

```
<ControlTemplate TargetType="Button">
  <Grid>
    <Rectangle x:Name="Rectangle1" Width="120" Height="120" Fill="Blue"/>
    <Rectangle x:Name="Rectangle2" Width="100" Height="100" Fill="LightBlue"/>
  </Grid>
</ControlTemplate>
```

The ControlTemplate is not the same thing at the actually rendered visuals in the visual tree. It is a blueprint which tells WPF how to create the VisualTree as shown in the diagram below

Figure 1Creating a ControlTemplate



For more information on Controls see the following document

Controls - Detail

Dependency Properties

Dependency Properties are arguably the most important abstraction in the whole of WPF. There are not many parts of the framework that would work without dependency properties. For example the following technologies all use depency properties

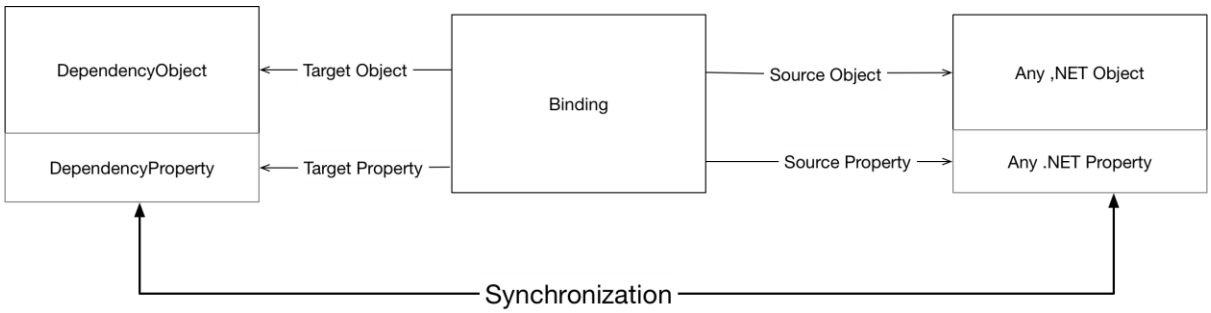
1. Data Binding
2. Property value inheritance / Sparse storage
3. Styles – only dependency properties can be styled
4. Property Triggers

For more information on dependency properties see DependencyProperties - Detail

Data Binding

A Binding keeps source and target properties in sync. The target property must be a dependency property and hence the target object must be a dependency object. The source property can be any .net property, however if we want the target to respond to changes in the source then the source object's type should implement `INotifyPropertyChanged`

Figure 2Binding Object



BINDING

❶ **Target Object**	Must be a DependencyObject
❷ **Target Property**	Must be DependencyProperty
❸ **Source Object**	Any .NET object but must implement INotifyPropertyChanged if we want change propagation

❹ **Source Property**	Any .NET property but again must raise INotifyPropertyChanged if we want change notification. Must also be a property and not just a field.

For more information on data binding see DataBinding - Detail

Styles and Triggers

A style is a collection of DependencyProperty setters that can be applied to multiple objects typically with the goal of providing a consistent appearance. Typically, styles are stored and accessed from ResourceDictionaries. Consider the following piece of XAML

```
<Window.Resources>
  <Style x:Key="StyleExample">
    <Setter Property="Shape.Fill" Value="Aqua"></Setter>
    <Setter Property="Rectangle.Width" Value="50"></Setter>
    <Setter Property="Rectangle.Height" Value="50"></Setter>
    <Setter Property="Rectangle.Margin" Value="10"></Setter>
  </Style>
</Window.Resources>
```

For more information on style and triggers see Styles - Detail

Layout

Layout is the process whereby each **`FrameworkElement`** is sized, positioned and rendered onto the screen. A parent **`Panel`** allocates a rectangular subset of its total available space to each child. The size and location of this rectangle defines a layout slot or bounding box onto which the child can be rendered. As a panel's children can also be panels, layout is implemented as a two-pass traversal of the element tree. In the first pass, known as measure, parents ask their children how much space they would like.

Inputs into process

- Available screen space
- Size of constraints
- Layout specific properties (margin and padding)
- Logic and behaviour of the parent panel

Layout occurs when an application starts and every time a window is resized.

For more information on layout see Layout - Detail

Rendering

From a high level, any UI technology is about processing user input from the mouse or keyboard and rendering the resultant state to the screen. Rendering in WPF is supported via the abstract base clas **`Visual`**. Visual is a lightweight implementation that supports

- Specifying and rendering drawing content
- Hit Testing
- Bounding box calculations
- Transformations
- Clipping

For more detail on low level renderinf see Rendering - detail

Vector Graphics

Vector graphic are used to render lines, paths and shapes to screen. The building blocks of vector graphics in WPF are Segments which are strung together end to end to form PathFigures. Multiple PathFigures can be combined to form a PathGeometry

For more information on vector graphics see

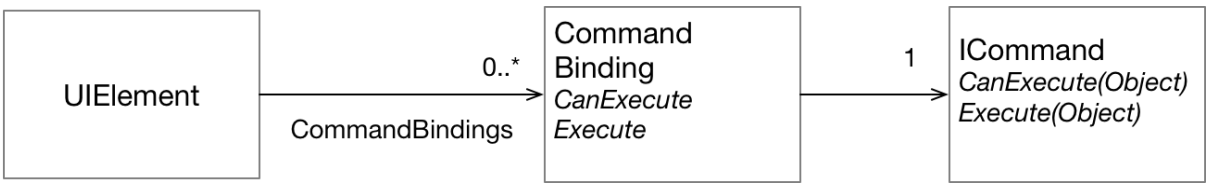
Vector Graphics - Detail

List Controls

Input – Commands and Events

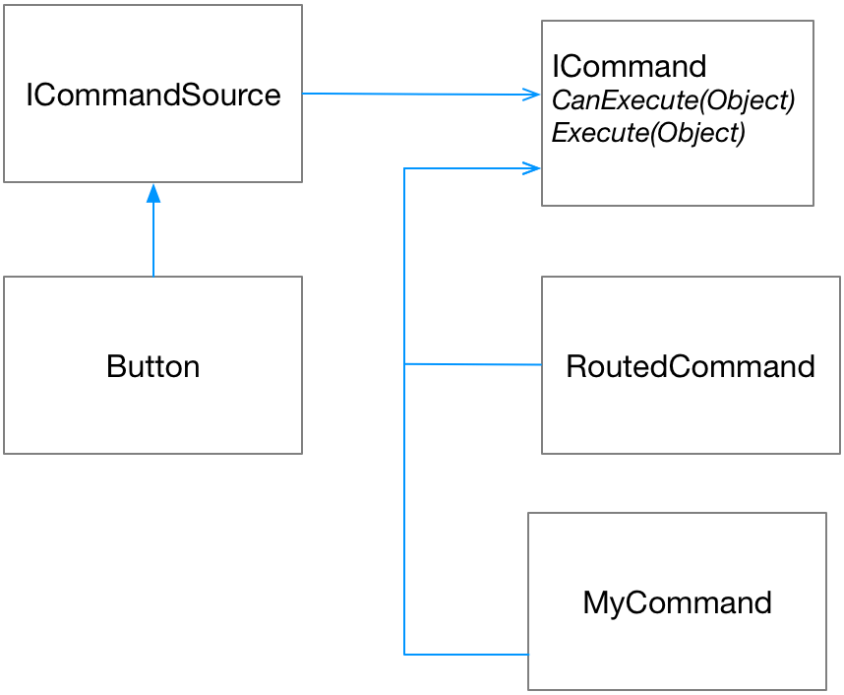
Command Binding

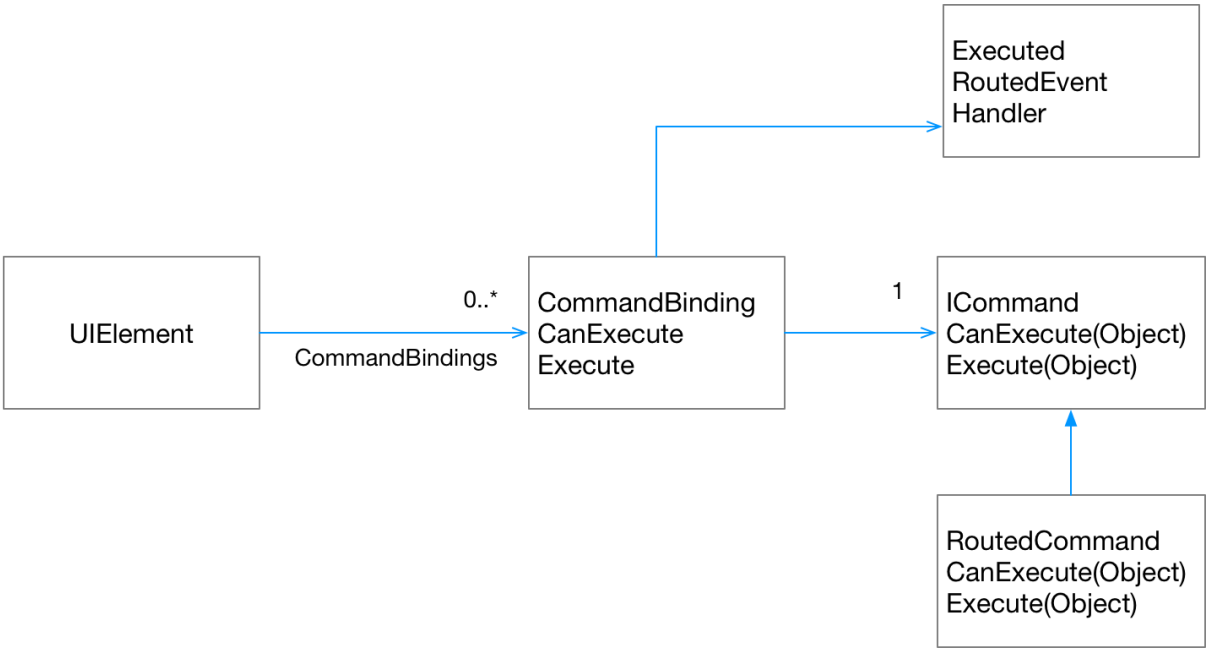
Each UIElement has a collection of CommandBinding objects. Each CommandBinding indicates that this UIElement can handle the binding's associated command.



RoutedCommand

RoutedCommand is the WPF implementation of ICommand. The execute method on a RoutedCommand does not directly contain the logic to carry out the command. Instead a RoutedCommand's Execute method raises an Execute event. The event is a RoutedEvent that bubbles/tunnels the element tree looking for a CommandBinding that knows how to handle that event. If it finds one it is a delegate attached to the CommandBinding that carries out the logic




Executed
RoutedEvent
Handler
UIElement
0..*
CommandBindings
CommandBinding
CanExecute
Execute
1
ICommand
CanExecute(Object)
Execute(Object)
RoutedCommand
CanExecute(Object)
Execute(Object)

```
    public class B_Visuals_Form_A_Hierarchy_A : Window
    {
        public B_Visuals_Form_A_Hierarchy_A()
        {
            // Dont show the title bar and border
            WindowStyle = WindowStyle.None;

            Background = Brushes.MidnightBlue;
            Foreground = Brushes.White;

            // Get rid of the resize parts
            ResizeMode = ResizeMode.NoResize;

            // Provide a means of dragging the window around
            MouseLeftButtonDown += (sender, args) => DragMove();

            // Set the window to have a square client are
            Height = 220;
            Width = 440;

            VisualCollection = new VisualCollection(this);
            VisualCollection.Add(CreateCircle());
            VisualCollection.Add(CreateSquare());
            ;
        }

        protected override int VisualChildrenCount => VisualCollection.Count;

        public VisualCollection VisualCollection { get; set; }

        protected override void OnRender(DrawingContext drawingContext)
        {
            drawingContext.DrawRectangle(Background, null,
                new Rect(new Point(1, 1), new Size(500, 500)));
        }

        protected override Visual GetVisualChild(int index)
        {
            return VisualCollection[index];
        }

        public DrawingVisual CreateCircle()
        {
            var circle = new DrawingVisual();
            var drawingContext = circle.RenderOpen();
            drawingContext.DrawEllipse(Foreground, null, new Point(110, 110), 100,
100);
            drawingContext.Close();
            return circle;
        }

        public DrawingVisual CreateSquare()
        {
            var shape = new DrawingVisual();
            var drawingContext = shape.RenderOpen();
            drawingContext.DrawRectangle(Foreground, null, new Rect(new Point(230,
10), new Size(200, 200)));
            drawingContext.Close();
            return shape;
        }
    }
```