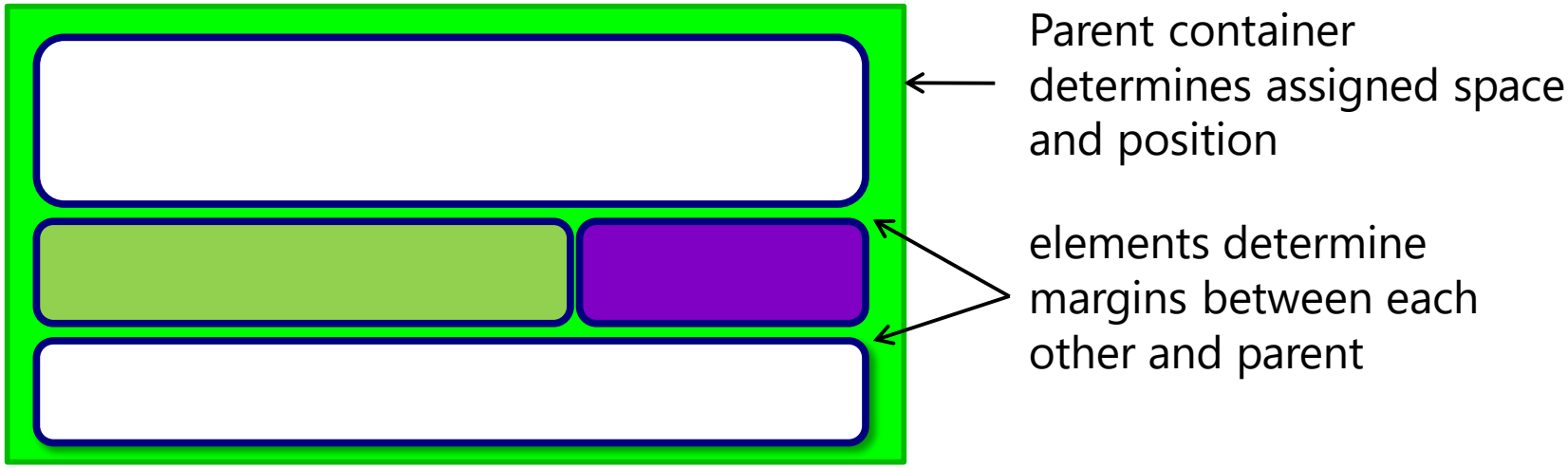
Managing Layout in WPF

DEVELOPMENTOR

DEVELOPING PEOPLE WHO DEVELOP SOFTWARE

Layout in WPF

- WPF manages layout through two mechanisms
  - visual elements request spacing and margins individually
  - containers enforce specific positions for child elements
- May seem cumbersome at first compared to other methods
  - but provides for more flexibility than just pixel-oriented layout



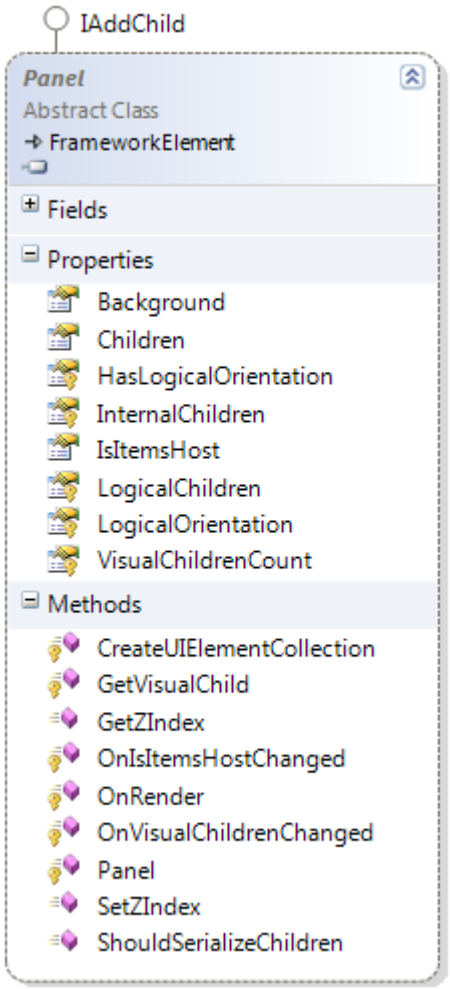
First stop - measurements

- WPF uses *device-independent pixels* to measure items
  - always specified as 1/96th of an inch
  - not affected by the screen resolution
- Always supplied as double values
  - allows fractional sizes
- Type convertors allow for units to be specified in XAML
  - units (default)
  - centimeters ("cm")
  - inches ("in")
  - font size points ("pt")

```
<Button Width="2in" Height="5cm" FontSize="24pt">
   This is a test
</Button>
```

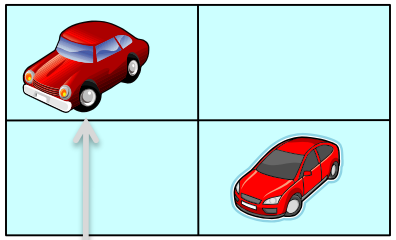
Second stop – the panel

- Panels organize content in WPF
  - each child is added to **`Children`** collection
  - children laid out based on the type of panel
  - panel decides size and position of each child
  - drawing order determined by position of child wit **`Panel.Children`** collection
- Panels can add background color but in gen affect visualization
  - they are intended to be a positioning container



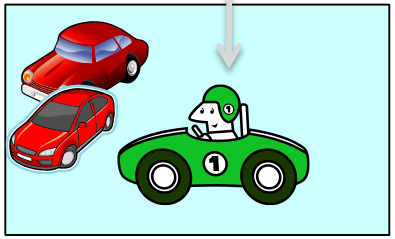
Layout mechanics

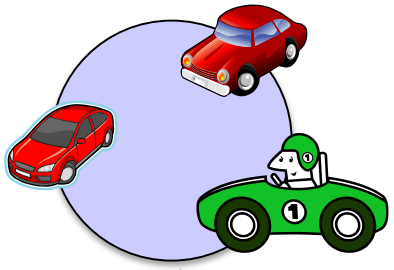
- Children must supply layout desires to panel
  - each panel likely has different layout rules and requirements



`Grid` needs to know `Column` and `Row`

`Canvas` needs to know `Top` and `Left` position





custom `RadialPanel` needs to know angle position

Supplying panel-specific layout information

- Panels define specific properties for layout management
  - values are "attached" to each child to indicate preferences
  - referred to as "attached properties" (more on this later)
  - **`Panel`** then reads current value from each child at runtime

```
Canvas panel = new Canvas();
Image sportsCar = LoadCarImage();
panel.Children.Add(sportsCar);

Canvas.SetLeft(sportsCar, 100.5);
Canvas.SetTop(sportsCar, 50.75);
```

static methods used to store value on `Image` in code

`Type.Property` syntax used in XAML

```
<Canvas>
<Image Source="sportsCar.jpg"
       Canvas.Top="50.75"
       Canvas.Left="100.5" />
</Canvas>
```

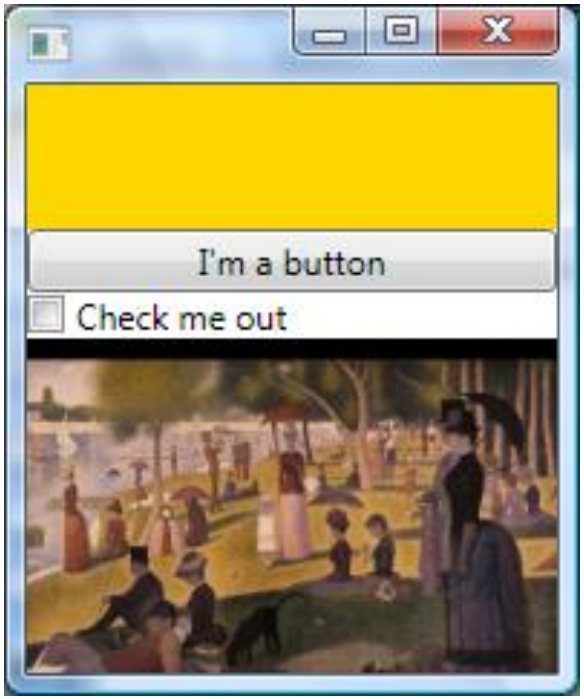
Built-in panels

- WPF includes support for most common layout scenarios
  - very common to compose panels together for complex layout

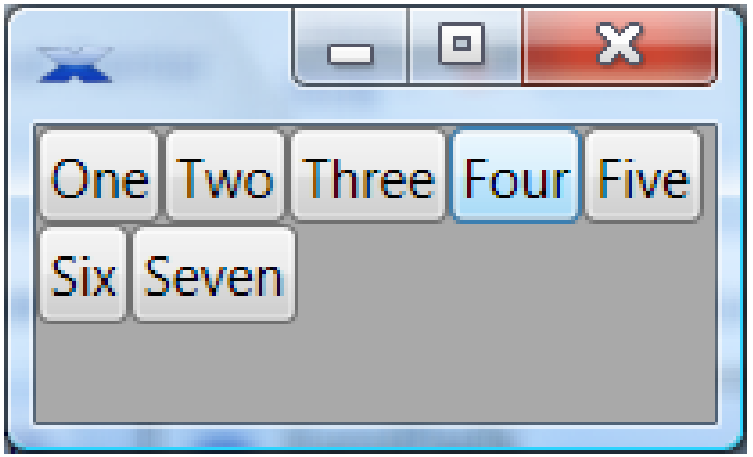
`StackPanel`	Organizes content horizontally or vertically in a stack
`WrapPanel`	Flows content around based on the size of the container
`Canvas`	Provides for pixel-based placement of children
`DockPanel`	Docks content to the edges of the parent
`UniformGrid`	Grid where all columns/rows are the same height/width
`Grid`	Grid where columns/rows can have different sizes

Stack-based layout

`StackPanel` lays out children horizontally or vertically



`WrapPanel` wraps children around available space horizontally or vertically



Notice how the default behavior is to pack in elements and use space as effectively as possible

Fine-tuning the position and spacing

- Properties on visual elements can fine-tune layout
  - allows children to *request* specific layout behavior

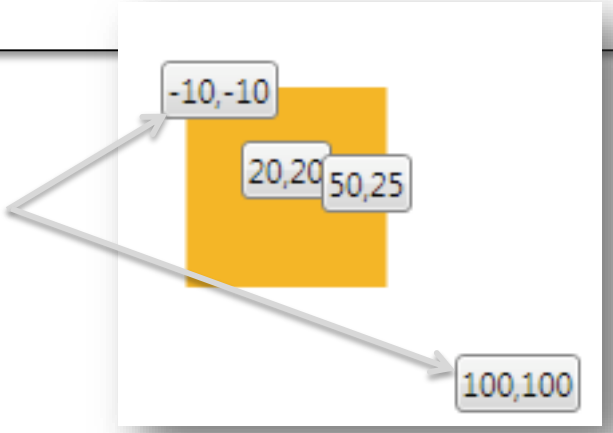
`HorizontalAlignment`	Horizontal position within parent
`VerticalAlignment`	Vertical position within parent
`Padding`	Spacing around content
`Margin`	Spacing around control
`MinHeight and MinWidth`[1]	Minimum width and height for element
`MaxHeight and MaxWidth`[1]	Maximum width and height for element
`Height`[2]	Requested height
`Width`[2]	Requested width
`HorizontalContentAlignment`	Horizontal position of the content
`VerticalContentAlignment`	Vertical position of content within control
`Panel.ZIndex`	Controls the Z-order positioning

Pixel-positioned layout

- Canvas provides traditional "location" based layout
  - children specify position using **`Canvas.SetXXXX`** methods or **`Canvas.XXXX`** attached property in XAML

```
<Canvas Background="Orange" Width="75" Height="75">
  <Button Canvas.Left="-10" Canvas.Top="-10" Content="-10,-10" />
  <Button Canvas.Left="20"  Canvas.Top="20"  Content="20,20" />
  <Button Canvas.Left="50"  Canvas.Top="25"  Content="50,25" />
  <Button Canvas.Left="100" Canvas.Top="100" Content="100,100" />
</Canvas>
```

Canvas allows "out-of-bounds" coordinates by default

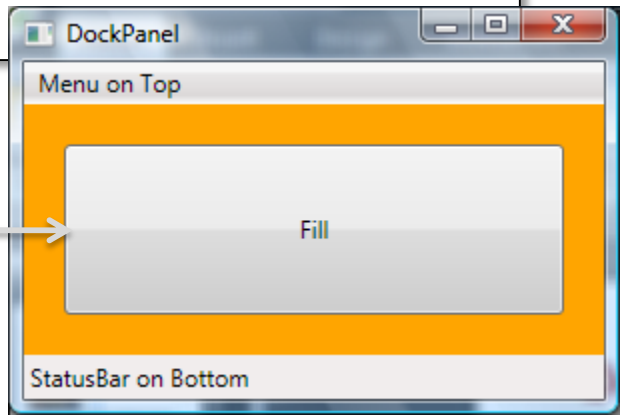


Docking elements

- DockPanel provides traditional "docking" behavior
  - child elements specify position using **DockPanel.SetDock** or attached property **DockPanel.Dock**

```
<DockPanel Background="Orange" LastChildFill="True">
   <Menu DockPanel.Dock="Top" />
   <StatusBar DockPanel.Dock="Bottom" />
   <Button Margin="20">Fill</Button>
</DockPanel>
```

final element "fills" remaining space by default – can be changed through `LastChildFill` property

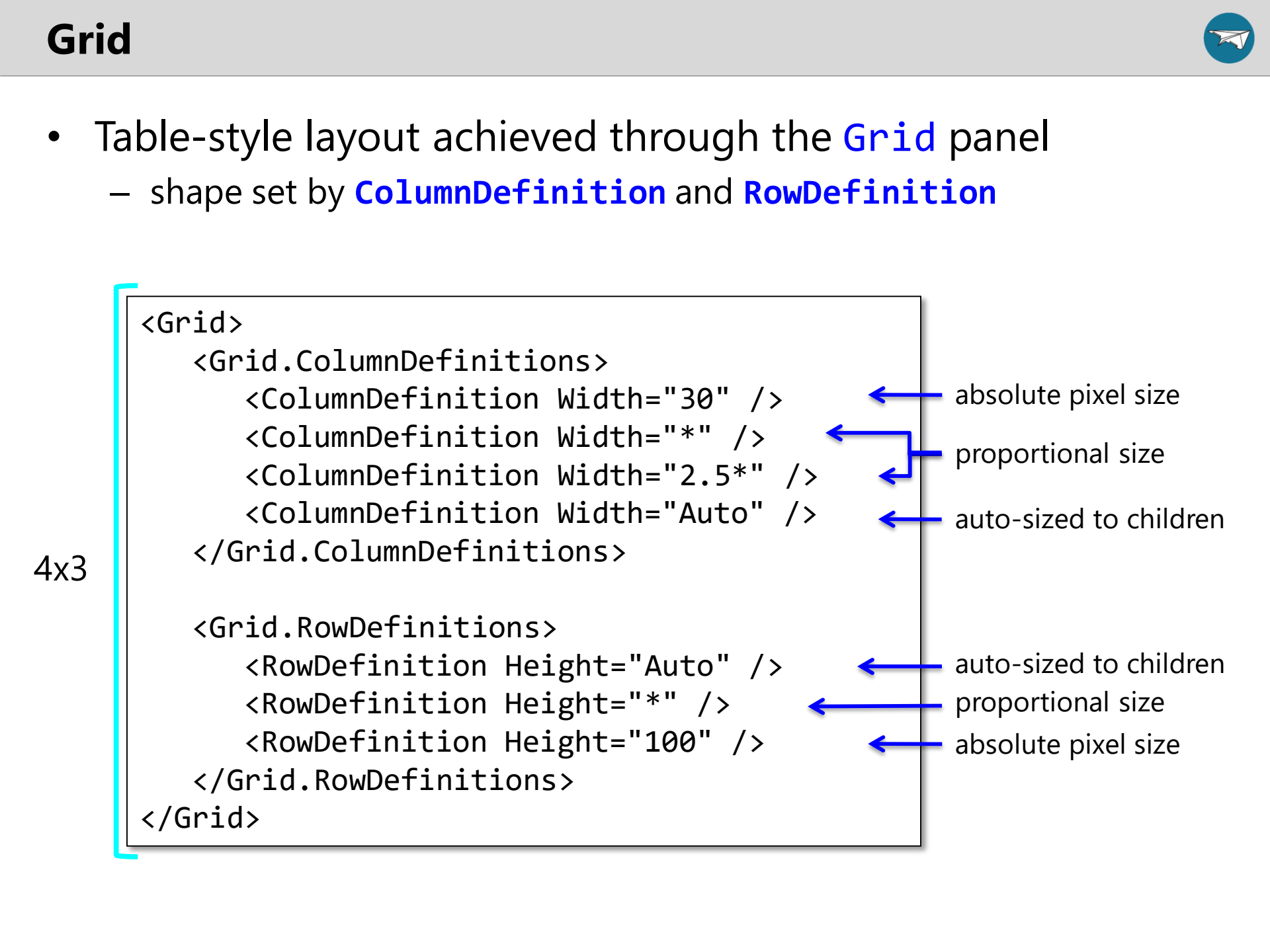


Grid

- Table-style layout achieved through the `Grid` panel
  - shape set by **`ColumnDefinition`** and **`RowDefinition`**

4x3

```
<Grid>
   <Grid.ColumnDefinitions>
      <ColumnDefinition Width="30" />
      <ColumnDefinition Width="*" />
      <ColumnDefinition Width="2.5*" />
      <ColumnDefinition Width="Auto" />
   </Grid.ColumnDefinitions>

   <Grid.RowDefinitions>
      <RowDefinition Height="Auto" />
      <RowDefinition Height="*" />
      <RowDefinition Height="100" />
   </Grid.RowDefinitions>
</Grid>
```



Positioning children in Grid

- Children can indicate row and column, defaults to [0,0]
  - using **Grid.Column** and **Grid.Row** attached properties
  - span with **Grid.ColumnSpan** and **Grid.RowSpan**

```
<Grid>
  ...
   <Button>1</Button>
   <Button Grid.Column="1">2</Button>
   <Button Grid.Column="0" Grid.Row="1"
           Grid.ColumnSpan="2" Content="3" />

</Grid>
```

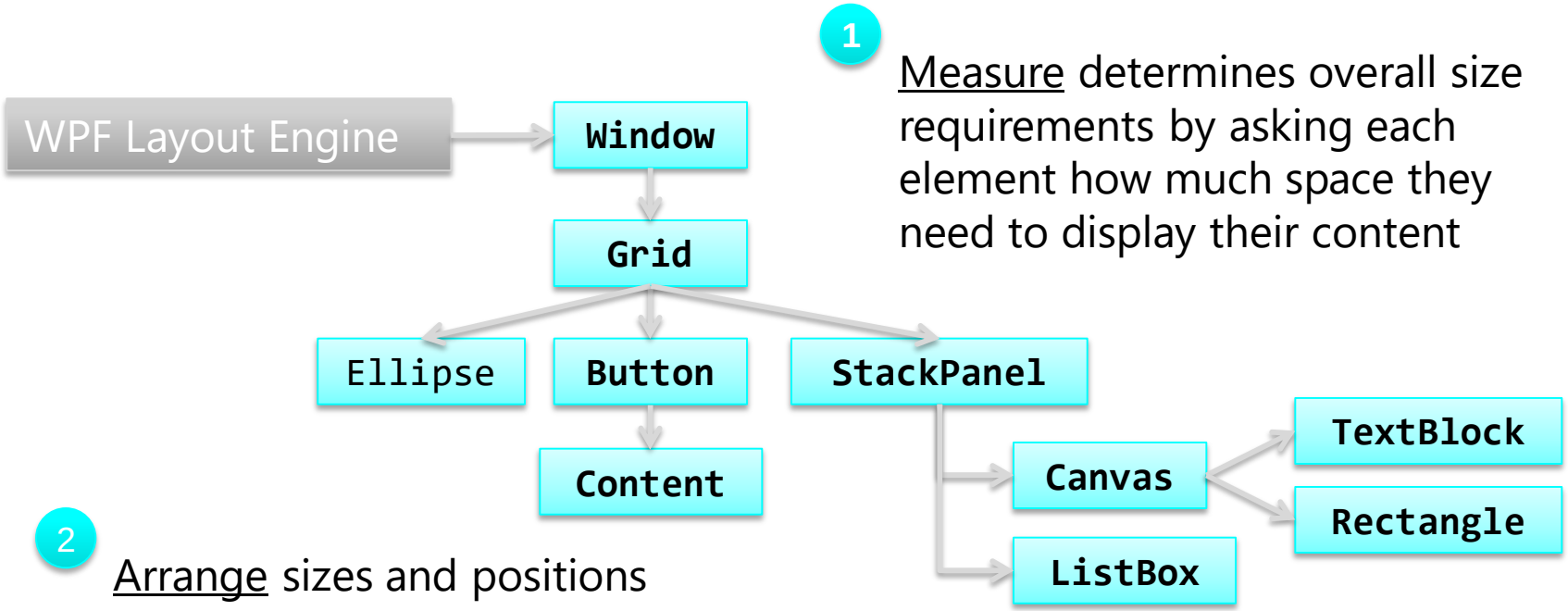
Dynamically resizing columns and rows

- Use `GridSplitter` to dynamically size rows and columns
  - best to give splitter own row or column
  - **`ResizeBehavior`** controls how resizing occurs

```
<Grid ShowGridLines="True">
   <Grid.ColumnDefinitions>
      <ColumnDefinition />
      <ColumnDefinition Width="Auto" />
      <ColumnDefinition />
   </Grid.ColumnDefinitions>

   <Button>1</Button>
   <GridSplitter Background="Black" Width="3" Grid.Column="1"
                 ResizeBehavior="PreviousAndNext" />
   <Button Grid.Column="2">2</Button>
</Grid>
```

Understanding the layout process

- WPF performs layout in a two-phase process that involves enumerating through all visual elements recursively

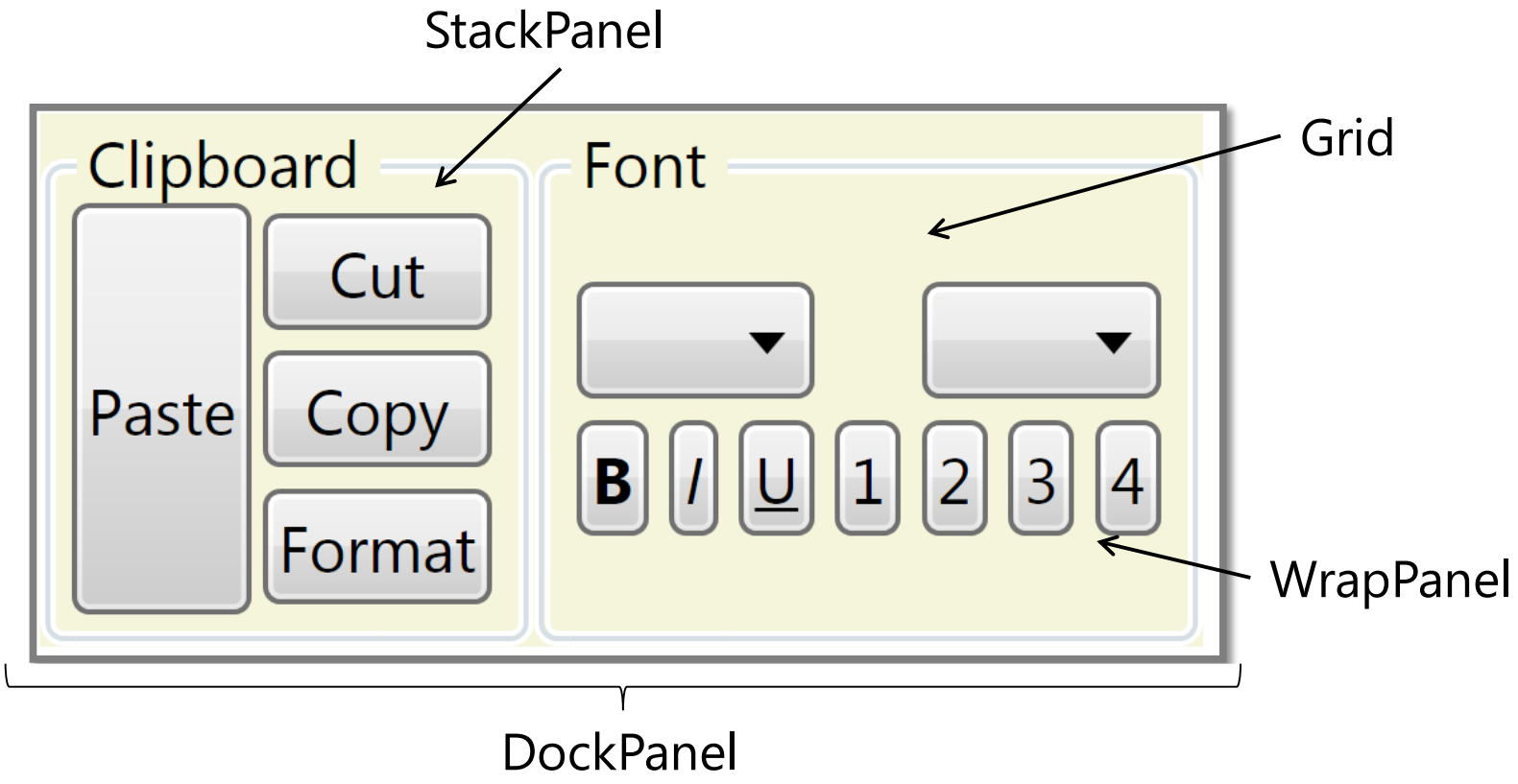


1 Measure determines overall size requirements by asking each element how much space they need to display their content

2 Arrange sizes and positions children based on measured requirements and available space

Putting it all together – building your UI

- Most UI designs will utilize multiple panels together
  - panels contained inside other panels



Choosing the appropriate panel

- `Grid` is the most common panel but also the most expensive
  - useful for table-style layout with fixed row and columns
- Choose a different panel if:
  - number of elements is dynamic
  - structure is simple or not tabular
  - performance is suffering due to large number of cells in **`Grid`**
- Remember panel composition can also solve many issues
  - breaking UI into "chunks" can make design more manageable
- If you are fighting the panel to get the proper layout:
  - consider using a different panel or some form of composition
  - write your own custom panel

Creating custom panels

- Useful to create custom panel if requirements are complex
  - … and you want to create a reusable component
- Custom panels derive from `Panel` base class[1]
  - override **`MeasureOverride`** to provide size information
  - override **`ArrangeOverride`** to layout children

CoverFlow style panel would be painful to repeat in code each time

Handling Overflow

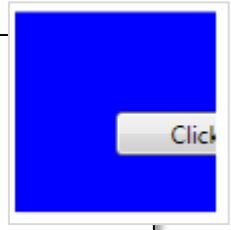
- Several strategies available when the content does not fit
  - clip the content to the bounds of the container
  - allow the content to be scrolled within the container
  - scale the content to fit the container
  - hide the content to make space

Clipping Content

- Clipping is the default way panels handle overflow
  - controlled through the **`ClipToBounds`** property
  - gets inherited down visual tree by all children
  - behavior is panel specific[1] and is ignored by most panels

```
<Canvas ClipToBounds="False" Background="Blue">
   <Button Canvas.Left="50" Canvas.Top="50"
           MinWidth="100">Click Me</Button>
</Canvas>
```

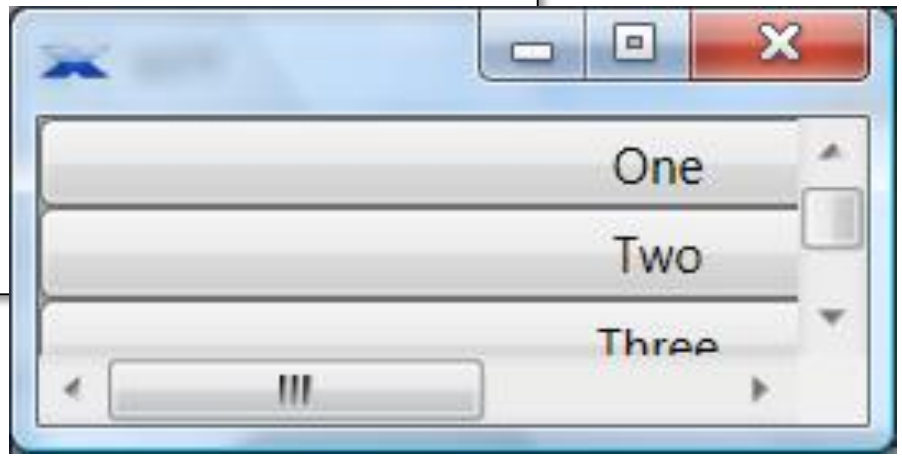

```
<Canvas ClipToBounds="True" Background="Blue">
   <Button Canvas.Left="50" Canvas.Top="50"
           MinWidth="100">Click Me</Button>
</Canvas>
```



Scrolling Content

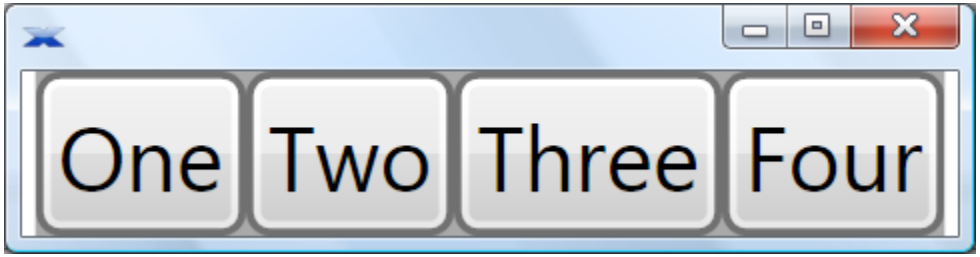
- `ScrollViewer` wraps content with scrollbars
  - **`HorizontalScrollBarVisibility`** and **`VerticalScrollBarVisibility`** control scroll bar usage
  - be careful about adding scrollable content inside **`ScrollViewer`**

```
<ScrollViewer HorizontalScrollBarVisibility="Auto"
              VerticalScrollBarVisibility="Auto">
   <StackPanel Background="DarkGray">
      <Button>One</Button>
      <Button>Two</Button>
      <Button>Three</Button>
      <Button>Four</Button>
   </StackPanel>
</ScrollViewer>
```



Scaling Content

- `Viewbox` scales content to fit the parent container
  - **`Stretch`** property controls how stretching will occur

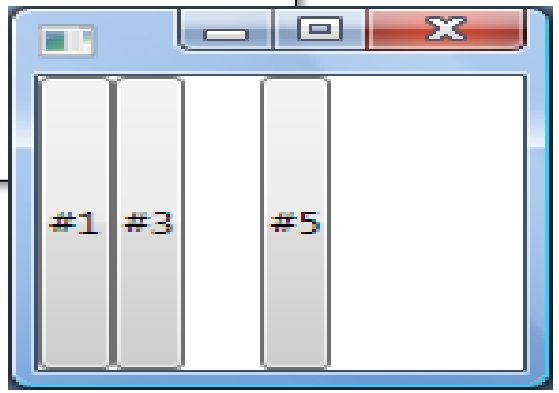



```
<Viewbox Stretch="Uniform">
   <StackPanel Orientation="Horizontal" Background="DarkGray">
      <Button>One</Button>
      <Button>Two</Button>
      <Button>Three</Button>
      <Button>Four</Button>
   </StackPanel>
</Viewbox>
```

Showing and Hiding content dynamically

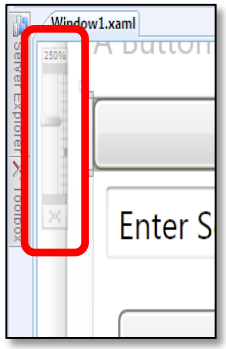
- Dynamic UI can be created using the `Visibility` property
  - elements can be **`Visible`**, **`Hidden`** or **`Collapsed`**

```
<StackPanel Orientation="Horizontal">

    <Button>#1</Button>
    <Button Visibility="Collapsed">#2</Button>
    <Button>#3</Button>
    <Button Visibility="Hidden">#4</Button>
    <Button>#5</Button>

</StackPanel>
```

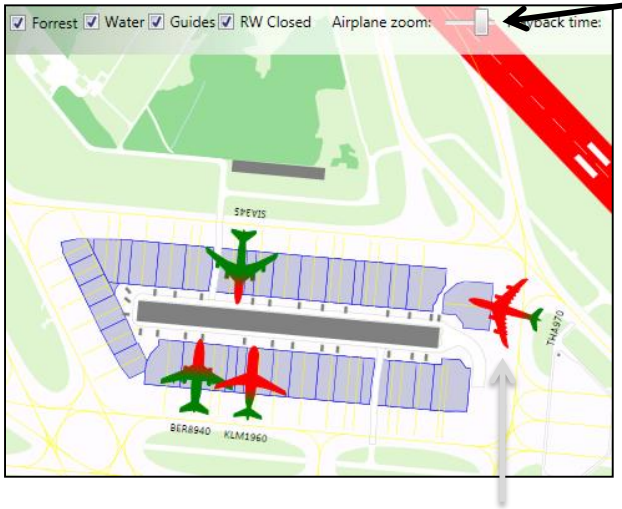


Dynamically changing layout at runtime

- Sometimes it is desirable to scale or reposition content
  - typically in response to some user activity (hover, click, etc.)

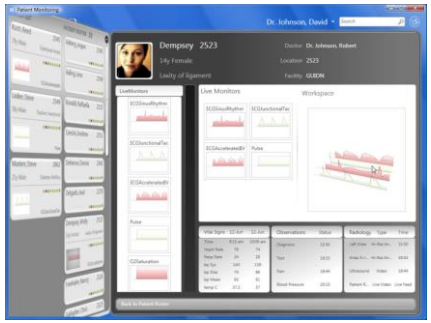


designer allows you to "zoom" into area for detail work



Airplane size controlled by slider

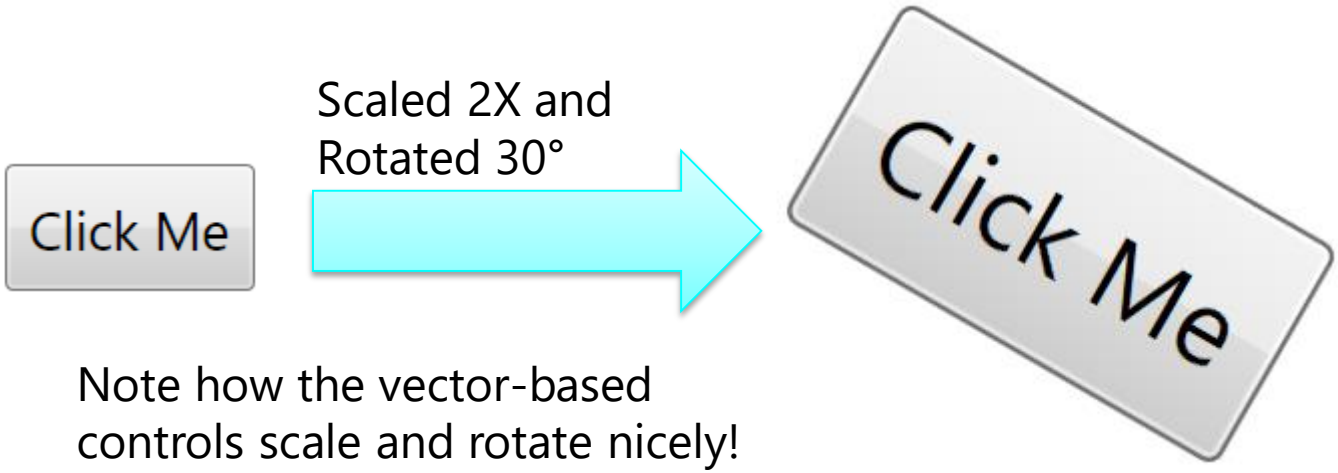
Airplane position can be altered through animation or mouse



patient card "flips" to reveal details when we click on it

Introducing Transforms

- Transformations can be applied to any `FrameworkElement`
  - becomes part of the drawing instructions for that element
  - available in two forms: **render** or **layout** transforms



Note how the vector-based controls scale and rotate nicely!

Render Transforms

- Render transforms are applied just prior to rendering
  - has no impact on other elements size or position
  - center of transform can be set by **RenderTransformOrigin**

```
<StackPanel>
   <Button Content="Click Me"
        RenderTransformOrigin=".5,.5">
      <Button.RenderTransform>
        ...
      </Button.RenderTransform>
   </Button>
   <Button>Button 1</Button>
</StackPanel>
```

origin is specified in relative coordinates [(0,0) – (1,1)]
(.5,.5) represents center of element – no matter what the real size is

Layout Transform

- Layout transforms are applied prior to layout
  - possibly affect positioning and size of elements
  - generally more expensive because it requires a layout pass


```
<StackPanel>
   <Button Content="Click Me">
      <Button.LayoutTransform>
            ...
      </Button.LayoutTransform>
   </Button>
   <Button>Button 1</Button>
</StackPanel>
```

second button is "pushed" down to make room for new size and rotation

Available transformations

reposition elements

```
<TranslateTransform X="50" Y="-50" />
```

```
<ScaleTransform ScaleX="2" ScaleY=".5"
                CenterX="50" CenterY="50" />
```

resize elements based on scale

rotate elements around center point

```
<RotateTransform Angle="90"
                 CenterX="50" CenterY="50" />
```

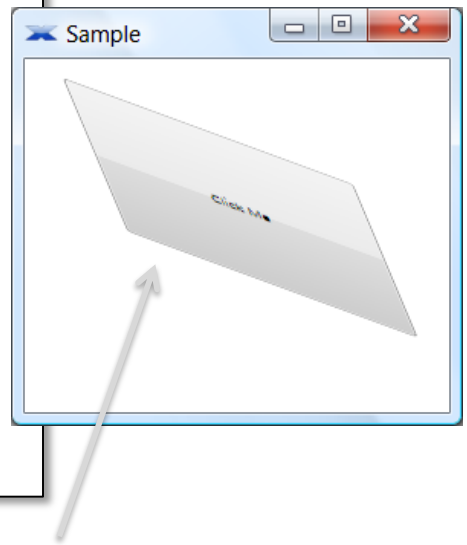
```
<SkewTransform AngleX="20" AngleY="10"
               CenterX="300" CenterY="50" />
```

skew elements around center point using separate angles

Applying multiple transforms together

- `TransformGroup` applies a group of individual transforms
  - transformed in the order specified

```
<Button Content="Click Me">
   <Button.RenderTransform>
      <TransformGroup>
        <ScaleTransform ScaleX=".5" ScaleY=".5" />
        <RotateTransform Angle="30" />
        <TranslateTransform X="10" Y="10" />
        <SkewTransform AngleX="45" />
      </TransformGroup>
   </Button.RenderTransform>
</Button>
```



Button can still be interacted with and is properly hit-testable

MatrixTransform

- Create custom 2D transformations with `MatrixTransform`
  - all other transforms are simple layers over this
  - **`Matrix`** property is a 3x3 affine transformation matrix

$$\begin{pmatrix} M11 & M12 & 0 \\ M21 & M22 & 0 \\ OffsetX & OffsetY & 1 \end{pmatrix}$$

OffsetX and OffsetY are `TranslateTransform`
M11 and M22 values used for scaling
Rotation and Skewing require angles calculated with cos/sin

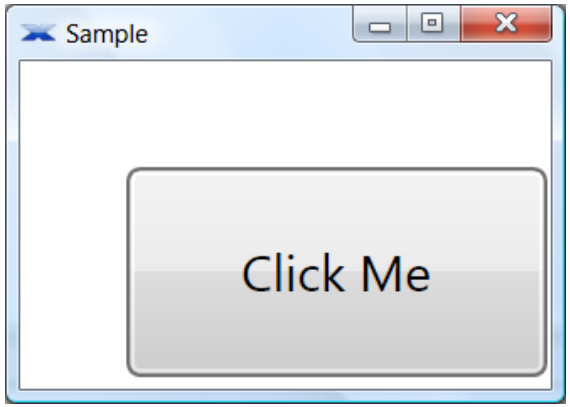
Using a MatrixTransform

- `Matrix` is specified as properties
  - or string syntax on **`RenderTransform/LayoutTransform`**

```
M11,M12, M21, M22, OffsetX, OffsetY
```

```
<Button Content="Click Me" Height="50" Width="100"
        RenderTransform="2, 0, 0, 2, 50, 50" />
```

Button scaled 2X and translated by 50 units



Summary

- Dynamic Layout was a main concern in WPF
  - prefer dynamic positioning
  - full complement of panels to manage any style of layout
  - most applications will combine different panels together
- Transforms are useful for repositioning and reshaping
  - no need to redraw or change code
  - very efficient and easy to use
- Consider adding no-op transforms for extensibility
  - allows easy animations in the future (more on this later)