Intro - Windows Presentation Foundation

Introduced in .Net 3.0 alongside:

● .. Communication Foundation (WCF)
  ○ SOAP / Web services
● .. Workflow Foundation (WWF)
  ○ Workflow Engine / Activities
● .. Presentation Foundation (WPF)
  ○ GUI framework

● Released in Nov '06
● Pre-installed on Windows Vista
Windows Forms (prev GUI framework)
- Pixel-based rendering with GDI+
- Not suited for 3D or video / animation
- IDE "...Designer.cs" files plus A LOT of code behind

Presentation Foundation (WPF)
- Rendered with DirectX, allows mid-pixel scaling to various resolutions
- First order support for animations
- Extensible Application Markup Language (XAML)
  - !!! Declarative syntax helps reduce code behind
Patterns - GUI Layers

GUIs are generally wrapped around existing code. The "top layer" in multi-layered systems.

GUI-specific patterns
  - Model View Controller (MVC)
  - Model View ViewModel (MVVM)

Supporting Patterns
  - Observer
Data Binding (Observer) - WPF

Instead of using code behind to set

\[
\text{myTextBox.Text} = \text{customer.CustomerName};
\]

Use declarative XAML

\[
\text{<TextBox Text="\{Binding CustomerName\}" />}
\]

GUI \textbf{observes} changes via \textbf{INotify}... interface
Data Binding programatically

Binding is a class, not just xml syntax

```
<TextBox Text="{Binding CustomerName}" />
```

.. done programatically, looks like

```c
Binding binding = new Binding("CustomerName");
textBoxCusto.SetBinding(TextBox.TextProperty, binding);
```

The "{Binding ..}" XAML is a MarkupExtension

- a kind of syntax "sugar"
- ex: {StaticResource ..} we'll see later in demo
public class Customer : INotifyPropertyChanged
{
    private string _customerName;
    public string CustomerName {
        set {
            _customerName = value;
            PropertyChanged(this, new PropertyChangedEventArgs("CustomerName"));
        }
    }
}

public event PropertyChangedEventHandler PropertyChanged;
**Binding Modes: Do it My Way**

**OneWay** Bindings: Source to Target (keeps GUI *current*)

**TwoWay** Bindings: Keeps GUI and backend *synchronized*

**UpdateSourceProperty** - controls how to update source:
- **LostFocus**: When user presses Enter, or Tabs away from TextBox
- **PropertyChanged**: After each Keystroke, backend is updated
Advanced Data Binding - WPF

Binding Converters augment bindings:
Ex: Show or Hide a control depending on a boolean
<Grid Visibility="{Binding IsDisabled,
Converter={StaticResource bool2VisibilityConverter}}" />

ex: Boolean.False $\rightarrow$ VisibilityMode.Hidden

MultiBindingConverters can combine multiple inputs and produce a single output
ex: Boolean.False + Status.Warning $\rightarrow$ Result.Continue

ValidationRules and IDataErrorInfo provide extensible mechanisms validate user input and notify the user.
Advanced Data Binding - WPF

ListCollectionView - allows List<T> to support SelectedItem

```xml
<ListView ItemsSource="{Binding Customers}"
    IsSynchronizedWithCurrentItem="True" />
```

codebehind can get currently selected item

```csharp
List<Customer> customers = ... //from somewhere
int selectedIndex = CollectionViewSource
```

- A lot of magic going on under the hood here

*REF: Bea Stollnitz's Blog (ex-Microsoft employee, now running Zag Studios)*

  ○ google "CollectionView wpf" #2 hit (best data binding info on web)
Taking a step back - code quality

Data Binding Pros:
- Decouples model classes from codebehind
- Greatly reduces size of codebehind
- XAML is more reusable than codebehind
- GUI is tolerant of failed bindings

Data Binding Cons:
- Since Bindings are established at runtime, failed bindings are not found until runtime.
  - ex: {Binding misspelled_PropertName}
  - Debug TraceLevel helps diagnose failures
Model / View / ViewModel (MVVM)

Views = GUI layouts / dialogs

ViewModels support each view; use models

Models = data classes

Key Benefits:
Centralizes View-Support code; allows Data Models to be GUI-agnostic; Testable.
WPF & Design Patterns

Design Patterns are general solutions to common problems that the language does not solve for you.

WPFs use of data-binding, extensive Style and ControlTemplate APIs and platform-like support of the MVVM pattern help solve some problems, pre-pattern.

now to Styles, and Animations (StoryBoards)
Imagine if we had to add Circle to this hierarchy!

Specialization can lead to bad designs like this

SpinningBlinking.. Problem
WPF Spinning Blinking Styles

Instead of having to specialize each control

WPF Allows us to
- Establish Transformations
- Animate Properties (ex: Opacity)
- Animated Transforms (ex: Angle)

And Styles make it applicable to any FrameworkElement
UIElement and FrameworkElement are very high in the hierarchy.

All Controls are FrameworkElement; support Transforms and Animations.
DirectX - Enabling Transforms

In WindowsForms, any UserDraw control was responsible for drawing pixels.

In WPF, DirectX can easily manipulate the drawing before rasterizing it to the screen.

Common Transforms:
- Rotate Transform
- SkewTransform
- ScaleTransform
Spinning: start with the transform

<Rectangle>
    <Rectangle.RenderTransform>
        <RotateTransform Angle="23" />
    </Rectangle.RenderTransform>
</Rectangle>

<Storyboard x:Key="spinningStoryboard">
    <DoubleAnimation Storyboard.TargetProperty="RenderTransform.Angle"
        From="0" To="360" Duration="0:0:5"
        RepeatBehavior="Forever"/>
</Storyboard>

when attached, we'll animate the Angle property (i.e spinning)
Transform transform = new RotateTransform();
rectangle.RenderTransform = transform;

//define the animation duration, range, etc.
var spinningAnimation = new DoubleAnimation(0, 360, new Duration(TimeSpan.FromSeconds(2)));

//Tell the RotateTransform to begin an animation
transform.BeginAnimation(RotateTransform.AngleProperty, spinningAnimation);

Abstraction: The Transform supports animation
Most WPF developers play a game (challenge)

- All XAML, no codebehind

We can trigger spinning on the `Loaded` event

```xml
<Rectangle.Triggers>
    <EventTrigger RoutedEvent="FrameworkElement.Loaded">
        <BeginStoryboard>
            <Storyboard>
                <DoubleAnimation Storyboard.TargetProperty="RenderTransform.Angle"
                                From="0" To="360" Duration="0:0:5"/>
            </Storyboard>
        </BeginStoryboard>
    </EventTrigger>
</Rectangle.Triggers>
```
Write the animation once, and put it in a Style

```xml
<Style x:Key="SpinningStyle">
    <Setter Property="UIElement.RenderTransform" Value="{StaticResource rotateTransform}" />
    <Style.Triggers>
        <EventTrigger RoutedEvent="FrameworkElement.Loaded">
            <BeginStoryboard Storyboard="{StaticResource spinningStoryboard}" />
        </EventTrigger>
    </Style.Triggers>
</Style>
```

Give it a name, so we can find it later.

RotateTransform was already defined as Resource, just re-use it.
ReUse / ReCycle (code)

Now we can apply that *spinningStyle* to any FrameworkElement.

And we can build other styles upon it

```xml
<Style x:Key="BlinkingSpinningStyle"
      BasedOn="{StaticResource SpinningStyle}"
>
  <Style.Triggers>
    <EventTrigger RoutedEvent="FrameworkElement.Loaded">
      <BeginStoryboard Storyboard ="{StaticResource blinkingStoryboard}"
      </EventTrigger>
    </Style.Triggers>
  </Style>
```
Blinking Style - Animates Opacity

UIElements also support Animating properties:
  o Opacity, Color, Width, Margins, etc.

```
<Style x:Key="BlinkingStyle">
...

  <BeginStoryboard><Storyboard>
    <DoubleAnimation
      Storyboard.TargetProperty="Opacity"
      From="1.1" To="0.1"
      Duration="0:0:0.8" AutoReverse="True" />

  </Storyboard></BeginStoryboard>
...

</Style>
```
SpinningBlinking...

It moves... (download code first)
Other Topics

You can create composite custom controls, that derive from UserControl.

- Adding DependencyProperties allows Binding
- AttachedProperties allow you influence parent controls

High Level Shader Language (HLSL)

- DirectX architecture allows some impressive 2D/3D effects
UserControl Example from Day Job

Single UserControl (circled in Red): bars across bottom have color and size data-bound to backend model.

ToggleButtons (circled in Green) are TwoWay bound to UserControl DependencyProperties

This graph is re-used in 3 different applications at my work.
End

More Info:
http://www.zagstudio.com/blog

Code for Download
https://docs.google.com/open?id=0B-7GE2fNRs7SRjZfVUpTQ1p2VjA

ellisda@gmail.com