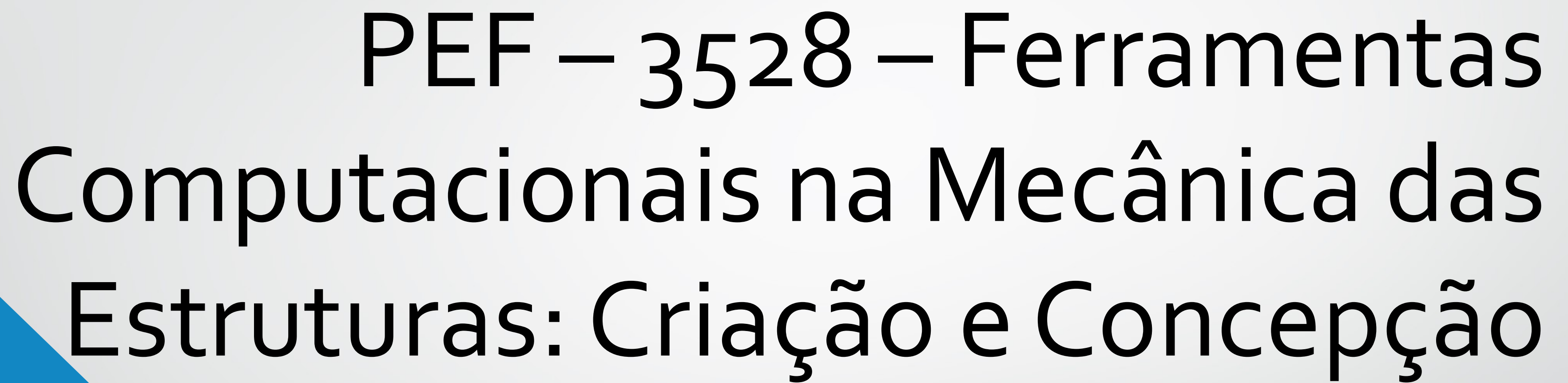
PEF – 3528 – Ferramentas Computacionais na Mecânica das Estruturas: Criação e Concepção



Prof. Dr. Rodrigo Provasi

e-mail: provasi@usp.br

Sala 09 – LEM – Prédio de Engenharia Civil

WPF

WPF

- WPF é um acrônimo para *Windows Presentation Foundation*.
- Veio para substituir a tecnologia de *Windows Forms.*

Características

- Maior integração com recursos gráficos 2D e 3D
- Independência de resolução
- Aceleração de *hardware*
- Programação declarativa → XAML
- Grande composição e customização
- Fácil *deployment*

Código

- É possível criar uma interface em WPF no C# apenas criando uma janela simples, como o código a seguir:

```
using System;
using System.Windows;

namespace Petzold.SayHello
{
        class SayHello
        {
            [STAThread]
            public static void Main()
            {
                Window win = new Window();
                win.Title = "Say Hello";
                win.Show();

                Application app = new Application();
                app.Run();
            }
        }
}
```

Código

- É possível atribuir um evento à janela:

```
using System;
using System.Windows;
using System.Windows.Input;

namespace Petzold.HandleAnEvent
{
  class HandleAnEvent
  {
    [STAThread]
    public static void Main()
    {
      Application app = new Application();
      Window win = new Window();
      win.Title = "Handle An Event";
      win.MouseDown += WindowOnMouseDown;

      app.Run(win);
    }
    static void WindowOnMouseDown(object sender, MouseButtonEventArgs args)
    {
      Window win = sender as Window;
      string strMessage = string.Format("Window clicked with {0} button at point ({1})",
            args.ChangedButton, args.GetPosition(win));
      MessageBox.Show(strMessage, win.Title);
    }
  }
}
```

Código

```
using System;
using System.Windows;
using System.Windows.Input;

namespace Petzold.GrowAndShrink
{
  public class GrowAndShrink : Window
  {
    [STAThread]
    public static void Main()
    {
      Application app = new Application();
      app.Run(new GrowAndShrink());
    }
```

```
public GrowAndShrink()
  {
    Title = "Grow & Shrink";      WindowStartupLocation = WindowStartupLocation.CenterScreen;
    Width = 192;      Height = 192;
  }
protected override void OnKeyDown (KeyEventArgs args)
  {
    base.OnKeyDown(args);

    if (args.Key == Key.Up)
    {
      Left -= 0.05 * Width;          Top -= 0.05 * Height;
      Width *= 1.1;          Height *= 1.1;
    }
    else if (args.Key == Key.Down)
    {
      Left += 0.05 * (Width /= 1.1);          Top += 0.05 * (Height /= 1.1);
}}}}
```

Markup

- Applicações WPF podem ser construídas por código como anteriormente mas, em geral, utiliza-se um tipo de XLM diferente denominado *zammel (XAML)*.

- Sua estrutura é:

Markup

```
<Button Foreground="LightSeaGreen" FontSize="24pt">
  Hello, XAML!
</Button>
```

- Esse markup pode ser traduzido como o seguinte código em C#:

```
Button btn = new Button();
btn.Foreground = Brushes.LightSeaGreen;
btn.FontSize = 24;
btn.Content = "Hello, XAML!"
```

Markup

- Um programa escrito em XAML costuma ser organizado em painéis:

```
<StackPanel>
   <Button Foreground="LightSeaGreen" FontSize="24pt">   Hello, XAML!    </Button>
  <Ellipse Fill="Brown" Width="200" Height="100" />

   <Button>
     <Image Source="http://www.charlespetzold.com/PetzoldTattoo.jpg"  Stretch="None" />
   </Button>
</StackPanel>
```

Markup

- Assim, é possível criar o seguinte *markup*:

```
<Button xmlns=”http://schemas.microsoft.com/winfx/2006/xaml/presentation”
    Content=”OK” Click=”button_Click”/>
```

- No C# fica:

```
System.Windows.Controls.Button b = new System.Windows.Controls.Button();
b.Click += new System.Windows.RoutedEventHandler(button_Click);
b.Content = “OK”;
```

Namespaces

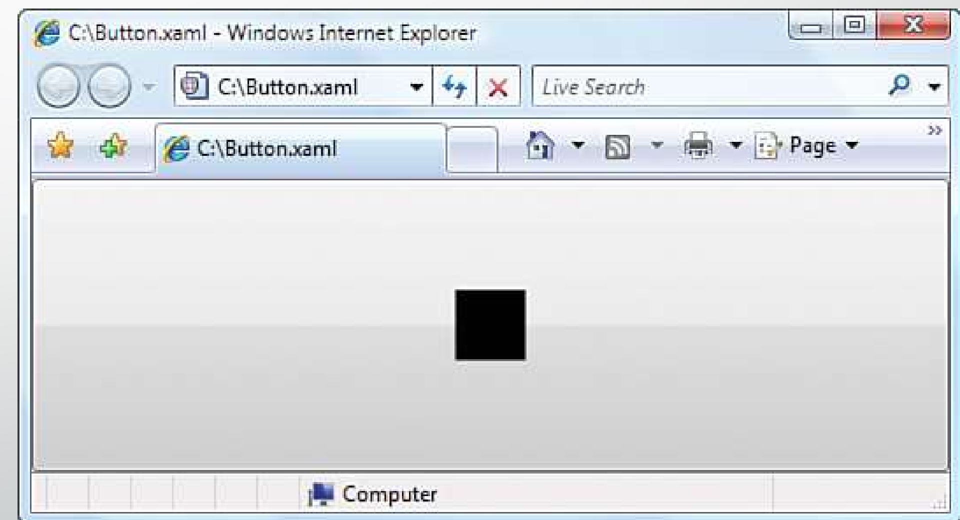
- Da mesma forma que precisa-se indicar os *namespaces* usados no C#, o mesmo procedimento é necessário no XAML.

- Para usar os elementos padrões do WPF utiliza-se o seguinte esquema (especificado no elemento na raiz do arquivo XAML):

xmlns=”http://schemas.microsoft.com/winfx/2006/xaml/presentation”

Propriedades

- As propriedades de um elemento em XAML são *strings* como vistas anteriormente, porém, como fazer se se deseja adicionar um retângulo à um botão (como se fosse um botão de stop de um aparelho de som antigo?)



Propriedades

- A solução é incluir *property elements* no XAML:

```
<Button xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation">
    <Button.Content>
        <Rectangle Height="40" Width="40" Fill="Black"/>
    </Button.Content>
</Button>
```

Propriedades

- Obviamente que isso vale para qualquer propriedade. O código a seguir pode ser reescrito em XAML como:

```
<Button xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
Content="OK" Background="White"/>

<Button xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation">
    <Button.Content>
        OK
    </Button.Content>
    <Button.Background>
        White
    </Button.Background>
</Button>
```

Converters

- Observe o seguinte trecho em C#:

System.Windows.Controls.Button b = new System.Windows.Controls.Button();

b.Content = "OK";

b.Background = System.Windows.Media.Brushes.White;

Converters

- O mesmo trecho em XAML fica:

```
<Button xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
Content="OK">
    <Button.Background>
        <SolidColorBrush Color="White"/>
    </Button.Background>
</Button>
```

Converters

- Como o compilador entende o que se digita em uma string (*White*) em um *Brush*?
- Isso se dá porque o compilador utiliza um conversor! Existem conversores já implementados no XAML mas há maneiras de se definir conversores próprios para traduzir textos em propriedades do objeto.

Extensões

- Extensões do *markup* são expressões colocadas diretamente no XAML e interpretadas pelo compilador.
- Todo comando entre chaves ({ }) é tratado como uma extensão do *markup* e não como um *string*.

Extensões

- Observe o seguinte trecho de XAML:

Markup extension class

```
<Button xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Background="{x:Null}"
        Height="{x:Static SystemParameters.IconHeight}"
        Content="{Binding Path=Height, RelativeSource={RelativeSource Self}}"/>
```

Positional parameter

Named parameters

Extensões

- O primeiro identificador após as chaves são a classe de extensão usada. No código, elas normalmente são acompanhadas no nome de *Extension*. Por exemplo, a classe *NullExtension* no *markup* é representado apenas como *Null*.

- *Binding* não é uma classe de extensão (e portanto está no *namespace* padrão) e por isso não tem o prefixo *x:* (que indica que se está usando o *namespace* especificado em *x*).

Extensões

- Se a extensão é suportada, parâmetros são passados usando vírgulas.
- Parâmetros posicionais são tratados como *strings* a serem passadas para os construtores.
- *Named Parameters* permitem definir propriedades da extensão.

Extensões

- O XAML anterior pode ser entendido como o a seguir:

```
<Button
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
    <Button.Background>
        <x:Null/>
    </Button.Background>
    <Button.Height>
        <x:Static Member="SystemParameters.IconHeight"/>
    </Button.Height>
    <Button.Content>
        <Binding Path="Height">
        <Binding.RelativeSource>
            <RelativeSource Mode="Self"/>
        </Binding.RelativeSource>
        </Binding>
    </Button.Content>
</Button>
```

Conteúdo

- Elementos podem conter elementos como seu conteúdo.
- Para compreender melhor esse processo, estudar-se-á como funciona o conteúdo de um controle em WPF.

Conteúdo

- Considere o seguinte código.

```
<Button xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
Content="OK"/>
```

- O mesmo pode ser reescrito como:

```
<Button xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation">
    OK
</Button>
```

Conteúdo

- Ou, usando um exemplo anterior:

```
<Button xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation">
    <Button.Content>
        <Rectangle Height="40" Width="40" Fill="Black"/>
    </Button.Content>
</Button>
```

- Esse código torna-se:

```
<Button xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation">
    <Rectangle Height="40" Width="40" Fill="Black"/>
</Button>
```

Conteúdo

- *Rectangle* é um objeto do WPF como outros controles.
- Sendo assim, pode-se colocar um controle dentro de outro e assim sucessivamente.
- Pode-se, por exemplo, ter um *combobox* ou um *listbox* com botões como itens.

Itens de coleções

- Existem vários controles em WPF que permitem coleções de itens e permitem a iteração nos elementos.

- São:
  - *Lists*
  - *Dictionaries*

Lists

- Listas são elementos como *array lists*. O WPF possui o controle ListBox que permite adicionar lista de elementos. O código a seguir adiciona 2 elementos à lista:

```
<ListBox xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation">
    <ListBox.Items>
        <ListBoxItem Content="Item 1"/>
        <ListBoxItem Content="Item 2"/>
    </ListBox.Items>
</ListBox>
```

Lists

- O código equivalente em C#:

```
System.Windows.Controls.ListBox listbox = new System.Windows.Controls.ListBox();
System.Windows.Controls.ListBoxItem item1 = new System.Windows.Controls.ListBoxItem();
System.Windows.Controls.ListBoxItem item2 =new System.Windows.Controls.ListBoxItem();
item1.Content = "Item 1";
item2.Content = "Item 2";
listbox.Items.Add(item1);
listbox.Items.Add(item2);
```

Dictionaries

- Dicionários criam pares de chaves e valores. Em geral são usados no WPF para criar dicionários de recursos (discutidos mais a frente). O código em XAML fica:

```
<ResourceDictionary xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
    <Color x:Key="1" A="255" R="255" G="255" B="255"/>
    <Color x:Key="2" A="0" R="0" G="0" B="0"/>
</ResourceDictionary>
```

Dictionaries

- O código equivalente em C#:

```
System.Windows.ResourceDictionary d = new System.Windows.ResourceDictionary();
System.Windows.Media.Color color1 = new System.Windows.Media.Color();
System.Windows.Media.Color color2 = new System.Windows.Media.Color();
color1.A = 255; color1.R = 255; color1.G = 255; color1.B = 255;
color2.A = 0; color2.R = 0; color2.G = 0; color2.B = 0;
d.Add("1", color1);
d.Add("2", color2);
```

Mesclando XAML e C#

- Para poder acessar uma propriedade do XAML em um outro objeto ou no código C# é necessário definir um nome para o objeto. Isso é feito da seguinte forma:

*<Button **x:Name="okButton"**>OK</Button>*

Mesclando XAML e C#

- Acessa-se o botão da seguinte forma:

```
Window window = null;
using (FileStream fs = new FileStream("MyWindow.xaml", FileMode.Open, FileAccess.Read))
{
    // Get the root element, which we know is a Window
    window = (Window)XamlReader.Load(fs);
}
// Grab the OK button, knowing only its name
Button okButton = (Button)window.FindName("okButton");
```

Mesclando XAML e C#

- Normalmente quando se cria uma janela no WPF, o compilador do VS cria um arquivo extra de C# com o mesmo nome. Esse arquivo é referido como *code behind*.
- Esse arquivo é uma declaração parcial da classe, ou seja, contém complementos a classe criada no XAML.

Mesclando XAML e C#

XAML

```
<Window
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
x:Class="MyNamespace.MyWindow">
...
</Window>
```

Code behind

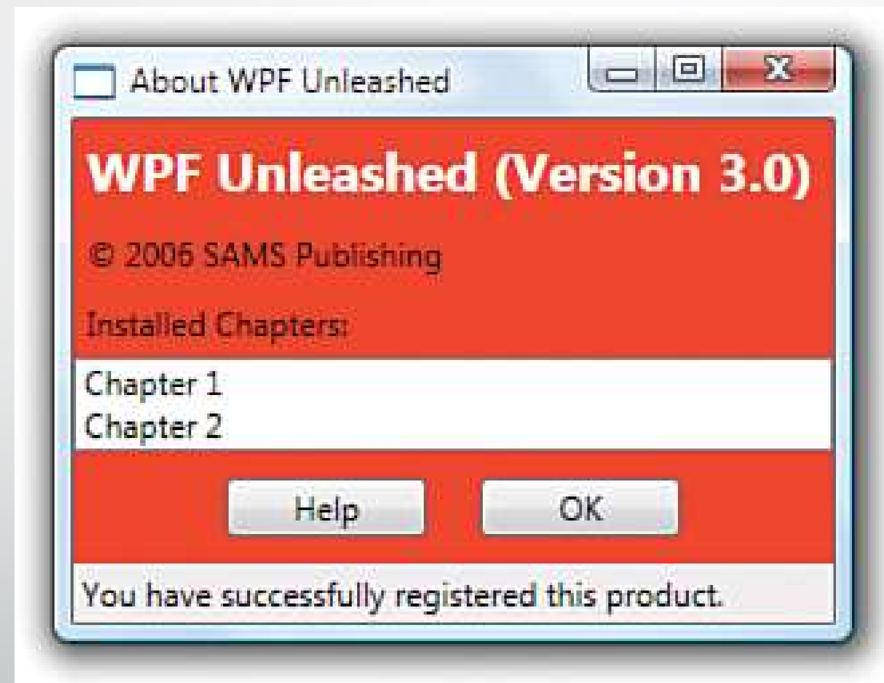
```
namespace MyNamespace
{
    partial class MyWindow : Window
    {
        public MyWindow
        {
            // Necessary to call in order to load XAML-defined content!
            InitializeComponent();
            ...
        }
        Any other members can go here...
    }
}
```

Estrutura e Eventos

Estruturas em WPF

```
<Window
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    Title="About WPF Unleashed" SizeToContent="WidthAndHeight"
    Background="OrangeRed">
    <StackPanel>
        <Label FontWeight="Bold" FontSize="20" Foreground="White">
            WPF Unleashed (Version 3.0)
        </Label>
        <Label>© 2006 SAMS Publishing</Label>
        <Label>Installed Chapters:</Label>
        <ListBox>
            <ListBoxItem>Chapter 1</ListBoxItem>
            <ListBoxItem>Chapter 2</ListBoxItem>
        </ListBox>
        <StackPanel Orientation="Horizontal" HorizontalAlignment="Center">
            <Button MinWidth="75" Margin="10">Help</Button>
            <Button MinWidth="75" Margin="10">OK</Button>
        </StackPanel>
    <StatusBar>You have successfully registered this product.</StatusBar>
    </StackPanel>
</Window>
```

Estruturas em WPF

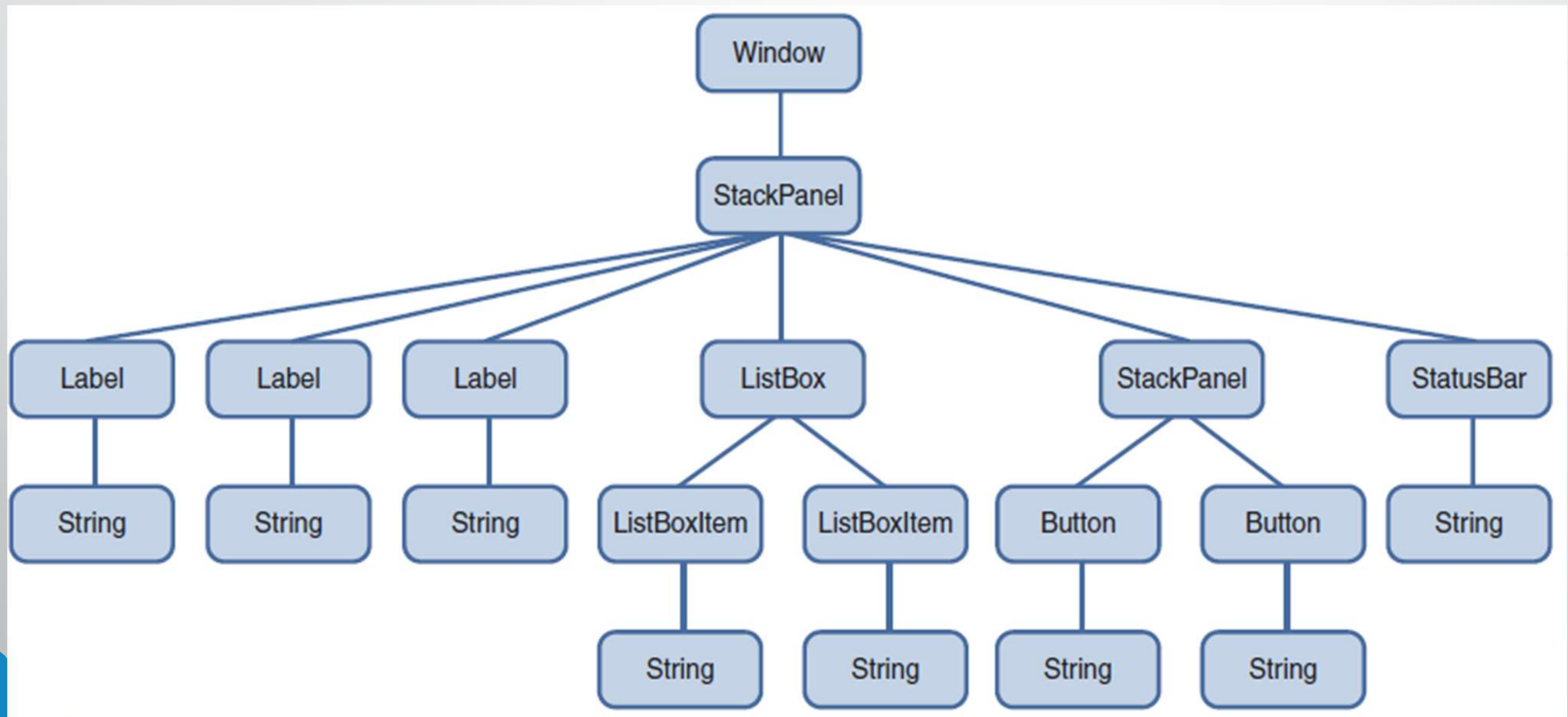


Estruturas em WPF

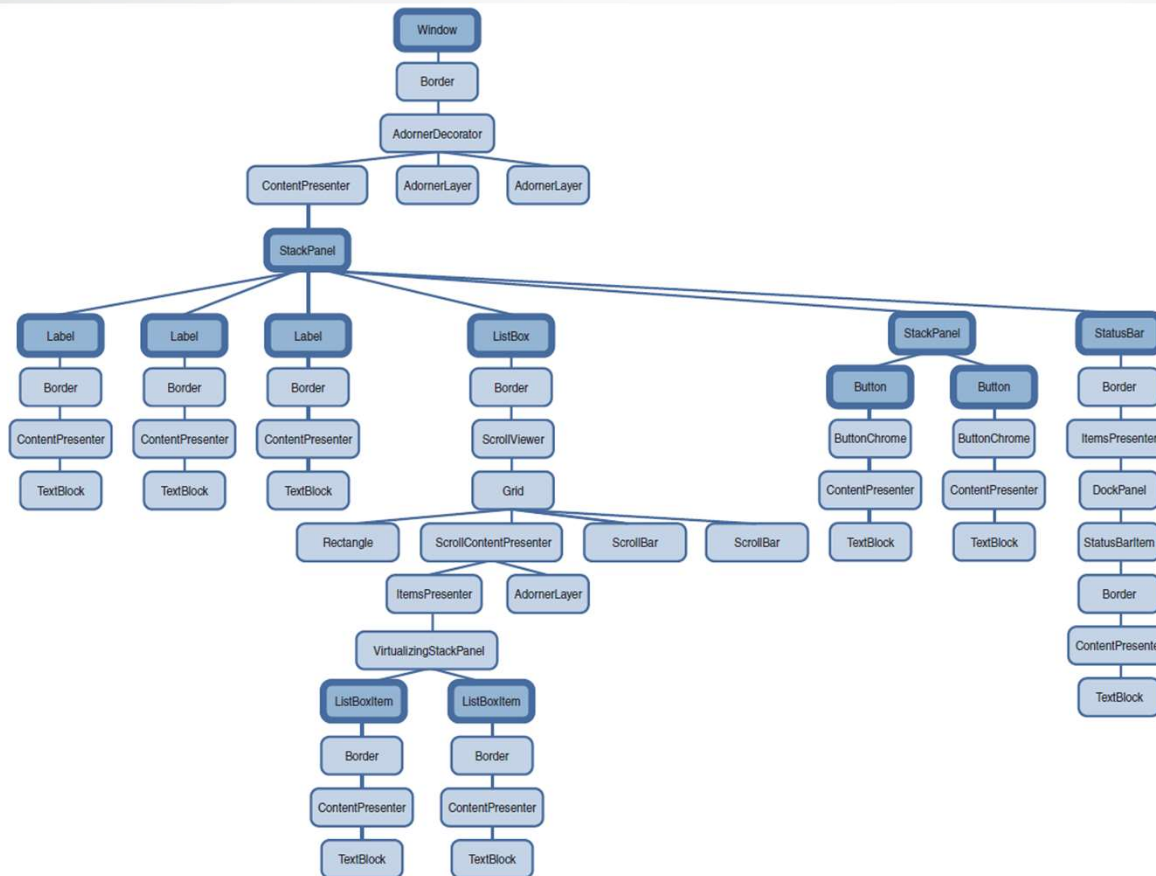
- O exemplo anterior possui uma árvore lógica. Isso é importante saber pois um evento que pode ser disparado pode ser propagado para vários elementos da árvore (para os filhos).

- Analogamente, há uma árvore visual, no qual elementos são colocados dentro de outros formando uma estrutura em árvore.

Estruturas em WPF



Estruturas em WPF



Dependency Properties

- O WPF utiliza um tipo de propriedade diferente da tradicional denominada *dependency property*.
- Esse tipo de propriedade permite uso em estilos, animações e *binding* de maneira simplificada.

Dependency Properties

- A implementação de uma propriedade dessa fica:

```
public class Button : ButtonBase
{
	// The dependency property
	public static readonly DependencyProperty IsDefaultProperty;
	static Button()
	{
		// Register the property
		Button.IsDefaultProperty = DependencyProperty.Register("IsDefault",
typeof(bool), typeof(Button), new FrameworkPropertyMetadata(false, new
PropertyChangedCallback(OnIsDefaultChanged)));
		…
	}
	// A .NET property wrapper (optional)
	public bool IsDefault
	{
		get { return (bool)GetValue(Button.IsDefaultProperty); }
		set { SetValue(Button.IsDefaultProperty, value); }
	}
	// A property changed callback (optional)
	private static void OnIsDefaultChanged(
	DependencyObject o, DependencyPropertyChangedEventArgs e) { … }
	…
}
```

Notificações

- Toda vez que uma *dependency property* muda seu valor, notificações são lançadas.

- Essas notificações são então transmitidas para os elementos e as atualizações necessárias na estrutura / interface são feitas.

Triggers

- Imagine o seguinte código:

*<Button **MouseEnter="Button_MouseEnter" MouseLeave="Button_MouseLeave"** MinWidth="75" Margin="10">Help</Button>*

*<Button **MouseEnter="Button_MouseEnter" MouseLeave="Button_MouseLeave"** MinWidth="75" Margin="10">OK</Button>*

Triggers

- Com o *code behind* a seguir:

```
// Change the foreground to blue when the mouse enters the button
void Button_MouseEnter(object sender, MouseEventArgs e)
{
    Button b = sender as Button;
    if (b != null) b.Foreground = Brushes.Blue;
}
// Restore the foreground to black when the mouse exits the button
void Button_MouseLeave(object sender, MouseEventArgs e)
{
    Button b = sender as Button;
    if (b != null) b.Foreground = Brushes.Black;
}
```

Triggers

- É possível fazer esse código usando um *trigger* (ativado pela mudança de alguma propriedade):

```
<Trigger Property="IsMouseOver" Value="True">
    <Setter Property="Foreground" Value="Blue"/>
</Trigger>
```

Trigger

- O único ponto que falta é dizer que esse *trigger* está relacionado ao botão.
- Isso é feito utilizando estilos.

Estilo

- Para o caso anterior:

```
<Button MinWidth="75" Margin="10">
    <Button.Style>
        <Style TargetType="{x:Type Button}">
            <Style.Triggers>
                <Trigger Property="IsMouseOver" Value="True">
                    <Setter Property="Foreground" Value="Blue"/>
                </Trigger>
            </Style.Triggers>
        </Style>
    </Button.Style>
    OK
</Button>
```

Routed Events

- *Routed Events* permitem que os eventos percorram os elementos da árvore visual mostrada anteriormente.
- A sua implementação é um pouco diferente de eventos tradicionais.
- A seguir como fica um exemplo:

Routed Events

```
public class Button : ButtonBase
{
    //The routed event
    public static readonly RoutedEvent ClickEvent;
    static Button()
    {
        // Register the event
        Button.ClickEvent =
EventManager.RegisterRoutedEvent("Click", RoutingStrategy.Bubble,
typeof(RoutedEventHandler), typeof(Button));
        …
    }
    // A .NET event wrapper (optional)
    public event RoutedEventHandler Click
    {
        add { AddHandler(Button.ClickEvent, value); }
        remove { RemoveHandler(Button.ClickEvent, value); }
    }
    protected override void
OnMouseLeftButtonDown(MouseButtonEventArgs e)
    {
        … // Raise the event
        RaiseEvent(new RoutedEventArgs(Button.ClickEvent, this)); …
    }
…
}
```

Routed Events

- Os eventos podem seguir vários caminhos:
  - **Tunneling** — O evento começa no elemento raiz e vai para os elementos mais baixos na hierarquia até que o elemento que chamou é atingido.
  - **Bubbling** — O evento começa no elemento que chamou e vai até que o elemento raiz seja atingido.
  - **Direct** — Somente o elemento que chamou o evento é atingido. Funciona como um evento tradicional em C#.

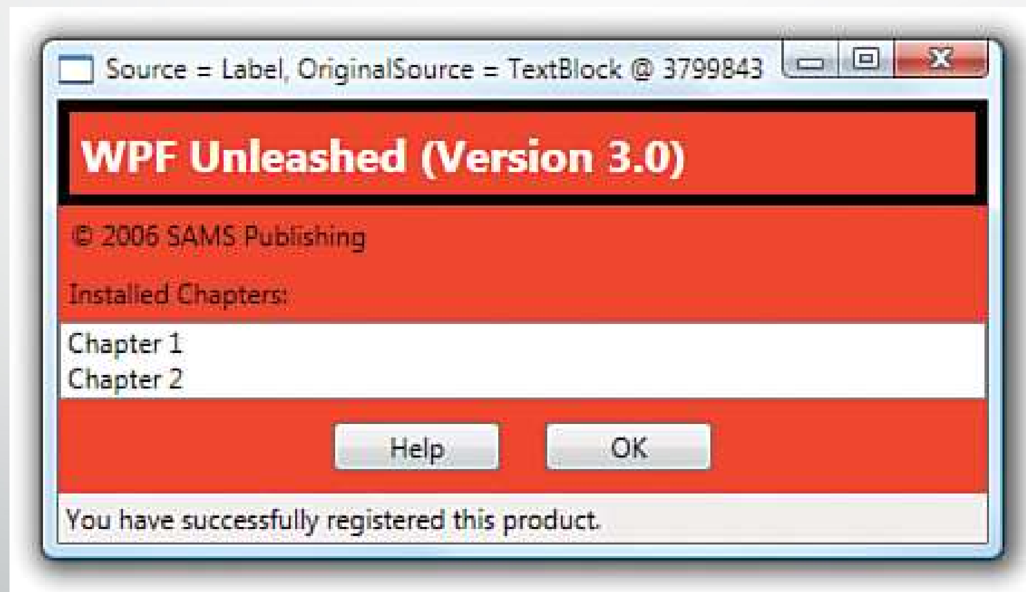
Routed Events

```
<Window
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
x:Class="AboutDialog"
MouseRightButtonDown="AboutDialog_MouseRightButtonDown"
Title="About WPF Unleashed" SizeToContent="WidthAndHeight"
Background="OrangeRed">
    <StackPanel>
        <Label FontWeight="Bold" FontSize="20"
Foreground="White">
            WPF Unleashed (Version 3.0)
        </Label>
        <Label>© 2006 SAMS Publishing</Label>
        <Label>Installed Chapters:</Label>
        <ListBox>
            <ListBoxItem>Chapter 1</ListBoxItem>
            <ListBoxItem>Chapter 2</ListBoxItem>
        </ListBox>
        <StackPanel Orientation="Horizontal"
HorizontalAlignment="Center">
            <Button MinWidth="75" Margin="10">Help</Button>
            <Button MinWidth="75" Margin="10">OK</Button>
        </StackPanel>
        <StatusBar>You have successfully registered this
product.</StatusBar>
    </StackPanel>
</Window>
```

Routed Events

```
using System.Windows;
using System.Windows.Input;
using System.Windows.Media;
using System.Windows.Controls;
public partial class AboutDialog : Window
{
        public AboutDialog()
        {
                InitializeComponent();
        }
        void AboutDialog_MouseRightButtonDown(object sender, MouseButtonEventArgs e)
        {
                // Display information about this event
                this.Title = "Source = " + e.Source.GetType().Name + ", OriginalSource = " +
                        e.OriginalSource.GetType().Name + " @ " + e.Timestamp;
                // In this example, all possible sources derive from Control
                Control source = e.Source as Control;
                // Toggle the border on the source control
                if (source.BorderThickness != new Thickness(5))
                {
                        source.BorderThickness = new Thickness(5);
                        source.BorderBrush = Brushes.Black;
                }
                else
                        source.BorderThickness = new Thickness(0);
        }
}
```

Routed Events



Commands

- Comandos são uma versão de eventos um pouco menos acoplada.
- Em geral são empregados em menus e botões de barras de ferramentas.

Commands

- A interface padrão do WPF permite três métodos a serem implementados em um Comando:
  - **Execute**: o método do comando propriamente dito.
  - **CanExecute**: função que retorna *true* se for possível executar o comando e *false* caso contrário.
  - **CanExecuteChanged**: método chamando toda vez que o *status* da função *CanExecute* mudar.

Commands

- Para implementar um comando, cria-se uma classe que implementa a interface *ICommand* e se cria um objeto dessa classe na janela em que ela for ser utilizada.

- Isso é bastante trabalhoso, porém o WPF possui implementações de comandos mais utilizados.

Commands

- Como dito, por padrão, o WPF implementa vários comandos:
  - **ApplicationCommands**—Close, Copy, Cut, Delete, Find, Help, New, Open, Paste, Print, PrintPreview, Properties, Redo, Replace, Save, SaveAs, SelectAll, Stop, Undo, and more
  - **ComponentCommands**—MoveDown, MoveLeft, MoveRight, MoveUp, ScrollByLine, ScrollPageDown, ScrollPageLeft, ScrollPageRight, ScrollPageUp, SelectToEnd, SelectToHome, SelectToPageDown, SelectToPageUp, and more
  - **MediaCommands**—ChannelDown, ChannelUp, DecreaseVolume, FastForward, IncreaseVolume, MuteVolume, NextTrack, Pause, Play, PreviousTrack, Record, Rewind, Select, Stop, and more

Commands

- **NavigationCommands**—BrowseBack, BrowseForward, BrowseHome, BrowseStop, Favorites, FirstPage, GoToPage, LastPage, NextPage, PreviousPage, Refresh, Search, Zoom, and more
- **EditingCommands**—AlignCenter, AlignJustify, AlignLeft, AlignRight, CorrectSpellingError, DecreaseFontSize, DecreaseIndentation, EnterLineBreak, EnterParagraphBreak, IgnoreSpellingError, IncreaseFontSize, IncreaseIndentation, MoveDownByLine, MoveDownByPage, MoveDownByParagraph, MoveLeftByCharacter, MoveLeftByWord, MoveRightByCharacter, MoveRightByWord, and more

Commands

- Assumindo que no exemplo o botão do *help* chama-se *helpButton*, pode-se escrever o seguinte código:

```
helpButton.Command = ApplicationCommands.Help;
helpButton.Content = ApplicationCommands.Help.Text;
```

Commands

- Usando o código agora, o botão ficará permanentemente desabilitado.

- Para resolver o problema, adiciona-se alguma lógica:

this.CommandBindings.Add(new CommandBinding(ApplicationCommands.Help, HelpExecuted, HelpCanExecute));

- E aí sim as com funções *HelpExecuted HelpCanExecute* implementadas, o botão funciona conforme o programado.