Guia preparatório para o exame 70-511
Windows Application development with Microsoft .NET Framework 4
Versão 1.0

Marcelo Sincic

TechEd 2010

Sessão CER-205
Dia 15/09/2010 as 9:00

Sumário

Prefácio 4
- Objetivo 4
- Sobre o Autor 4
- Links do Autor 4

Informações do Site de Certificação (original em inglês) 5
- Sobre o Exame 5
- Perfil do Candidato 5
- Certificação Alcançada 5
- Tópicos cobertos pelo exame 5
  - Building a User Interface by Using Basic Techniques (23%) 5
  - Enhancing a User Interface by Using Advanced Techniques (21%) 6
  - Managing Data at the User Interface Layer (23%) 6
  - Enhancing the Functionality and Usability of a Solution (17%) 7
  - Stabilizing and Releasing a Solution (17%) 7

Tópico 1 – Criando interfaces básicas 9
- Objetos de conteúdo 9
- Recursos estáticos e compartilhados (dicionários) 9
- Estilos 10
- Herdando e sobrescrevendo estilo 10
- Animações 10
- Temas 11
- Trigger de animação 12

Tópico 2 - Criando Interface de usuários com técnicas avançadas 14
- Trabalhando com eventos 14
- Associação com comandos 14
- Modificando a interface em runtime 15
- Templates 15
- Controles customizados (UserControls) 16
- Criando objetos gráficos 16
- Conteúdo multimídia 17
- Codificando e decodificando imagens (conversão) 17
- Utilizando imagens 17
- Menus 18

Tópico 3 - Trabalhando com dados (Data bindings) 19
- Implementando data bindings 19
- Binding com propriedades 19
- Binding com XML 20
- Binding com outros controles 21
- Notificações de alteração para objetos binding 21
- Binding com banco de dados 22

Binding com LINQ 22
Data Template 22
Conversão de dados para binding 23
BindingSource para WinForms 23
Mostrando ícones de validação 24
Erros em regras de negócio 24
Tratamento de erros não gerenciados na Aplicação 25
Tópico 4 – Usabilidade e funcionalidades avançadas 27
Integrando controles WPF com controles WinForm 27
Integrando controles WinForms com WPF 27
Processos assíncronos – Dispatcher 28
Processos assíncronos – ThreadPool 28
Processos assíncronos – BackgroundWorker 29
Processo assíncronos – TPL (Task Parallel Linq) 29
Localização 29
Globalização (processo manual) 29
Aplicativo LocBAML para localização 30
Drag-and-Drop 30
Segurança - Security Restriction Policy (SRP) 30
Segurança – Code Access Security (CAS) 30
Segurança – UAC 31
Segurança – Partial and Full Trust 31
Application and User Settings 32
Tópico 5 – Estabilização e Deploy 34
Testes 34
Unit Test 34
Teste de interface gráfica 35
Debug - IntelliTrace 35
Debug - WPF Visualizer 36
Debug - PresentationTraceSources 37
Deploy – XCOPY 37
Deploy - Windows Installer Project 37
Deploy – ClickOnce 38
Arquivo manifesto 38

Prefácio

Objetivo

Este manual foi criado com o objetivo de ajudar os interessados em fazer o exame 70-511 da Microsoft, em função das palestras da track Certificação do TechEd 2010.

Uma certificação hoje é importante para profissionais que querem se destacar dos demais em um processo de seleção e também para os novos demonstrando sua habilidade por meio de um teste executado pelo fabricante da tecnologia. A certificação não lhe garante um emprego, mas lhe dará a um entrevistador uma diferenciação.

Este manual não irá responder perguntas nem fornecer a base do treinamento oficial 10262 que a Microsoft vende por meio de seus parceiros de Learning Services. Mas a intenção é servir de um breve guia para quando você estiver estudando para o exame, para saber por onde começar.

Muitas das questões do exame de certificação são baseadas em cenários e não apenas em perguntas técnicas. Ao ler este manual tente imaginar o motivo de existir a feature e compare com as outras do mesmo tipo. Tente levar em conta cenários de empresas que utilizariam isso e como você faria.

Leia os links, muitos tópicos são complexos e não seria possível descrever sem escrever um livro.

Por fim, utilize o Visual Studio 2010 mesmo que em versão de avaliação para estudar, não confie apenas nas informações aqui e, principalmente, **não recorra a BrainDumps**, pois eles podem até te ajudar mas criarão em você uma falsa confiança que lhe será cobrada no futuro e denigrirá a imagem dos que passaram no exame honestamente, com muito esforço, como você está tentando agora.

Sobre o Autor

Marcelo Sincic é certificado Microsoft MCITP, MCPD, MCTS, MCSA, MCDBA, MCAD e MCT pela IBM como CLP Domino 6.5/7.0 e pela Sun como Java Trainer. É certificado em System Center OM/CM, Windows 7, Windows 2008, SQL Server 2000/2005/2008, Sharepoint 2007/2010, Forefront, ASP.NET 3.5, Windows Mobile e Hyper-V, recebendo o titulo Charter Member em diversas certificações. Recebeu premio como um dos 5 melhores instrutores da América Latina em 2009.
Atualmente é consultor e instrutor na plataforma Microsoft, mas é desenvolvedor desde 1988 com Clipper S'87 e Dbase III com Novell 2.0. Perfil completo em http://www.marcelosincic.com.br/blog/page/Sobre-o-Autor.aspx

Links do Autor

Blog: http://www.marcelosincic.com.br
Twitter: http://twitter.com/marcelosincic
LinkedIn: http://www.linkedin.com/pub/marcelo-sincic/1/271/39b
Facebook: http://www.facebook.com/msincic

Informações do Site de Certificação (original em inglês)

Sobre o Exame

This exam is designed to test the candidate's knowledge and skills for developing applications using Windows Forms, WPF and the .NET Framework 4.
Questions that contain code will be presented in either VB or C#. Candidates can select one of these languages when they start the exam.

Perfil do Candidato

The candidate works in a development environment that uses Microsoft Visual Studio .NET 2010 and Microsoft .NET Framework 4.0 to create WinForms and WPF applications. The candidate should have at least one year of experience developing Windows-based applications by using Visual Studio, including at least six months of experience with Visual Studio 2010 Professional.
In addition, the candidate should be able to demonstrate the following:

- a solid understanding of the .NET Framework 4.0 solution stack for WPF and WinForm applications
- experience in creating data-driven user interfaces for WPF and WinForm applications
- experience in creating layouts by using Extensible Application Markup Language (XAML)
- experience in programming against the WPF and WinForm object model
- experience with unit testing using MSTest
- experience with setup and deployment projects

Certificação Alcançada

When you pass Exam 70-511: TS: Windows Applications Development with Microsoft .NET Framework 4, you complete the requirements for the following certification(s):

- MCTS: .NET Framework 4, Windows Applications

Exam 70-511: TS: Windows Applications Development with Microsoft .NET Framework 4: counts as credit toward the following certification(s):

- MCPD: Windows Developer 4

Tópicos cobertos pelo exame

Building a User Interface by Using Basic Techniques (23%)

- Choose the most appropriate control class.

This objective may include but is not limited to: evaluating design requirements and then selecting the most appropriate control based on those requirements; recognizing when none of the standard controls meet requirements; item controls, menu controls, content controls
This objective does not include: designing a custom control

- Implement screen layout by using nested control hierarchies.

This objective may include but is not limited to: using panel-derived controls, attaching properties
This objective does not include: items controls, control customization

- Create and apply styles and theming.

This objective may include but is not limited to: application-level styles, overriding styles, style inheritance, Generic.xaml, theming attributes
This objective does not include: data-grid view style sharing

- Manage reusable resources.

This objective may include but is not limited to: fonts, styles, data sources, images, resource dictionaries, resource-only DLLs

- Implement an animation in WPF.

This objective may include but is not limited to: creating a storyboard; controlling timelines; controlling the behavior when the animation completes; double, color, and point animations; starting an animation from code and from XAML

This objective does not include: direct rendering updates, implementing key frame animations

Enhancing a User Interface by Using Advanced Techniques (21%)

•Manage routed events in WPF.

This objective may include but is not limited to: tunneling vs. bubbling events, handling and cancelling events

This objective does not include: simple event handling; creating custom events

•Configure WPF commanding.

This objective may include but is not limited to: defining WPF commands based on RoutedCommand; associating commands to controls; handling commands; command bindings; input gestures

This objective does not include: creating custom commands by implementing ICommand

•Modify the visual interface at run time.

This objective may include but is not limited to: adding/removing controls at run time; manipulating the visual tree; control life cycle; generating a template dynamically

This objective does not include: instantiating forms and simple modification of control properties at runtime

•Implement user-defined controls.

This objective may include but is not limited to: deciding whether to use a user/composite, extended, or custom control ; creating a user/composite control; extending from an existing control

This objective does not include: creating a custom control by inheriting directly from the Control class and writing code

•Create and display graphics.

This objective may include but is not limited to: creating and displaying graphics by using geometric transformation; brushes; drawing shapes; clipping; double buffering; overriding Render (WPF) and OnPaint (WinForms); differentiating between retained and non-retained graphics

This objective does not include: creating and displaying three-dimensional graphics; hit testing; creating images

•Add multimedia content to an application in WPF.

This objective may include but is not limited to: media player vs. media element; adding a sound player; images

This objective does not include: buffering

•Create and apply control templates in WPF.

This objective may include but is not limited to: template binding

This objective does not include: styling and theming; data templating

•Create data, event, and property triggers in WPF.

Managing Data at the User Interface Layer (23%)

•Implement data binding.

This objective may include but is not limited to: binding options, static and dynamic resources, element bindings, setting the correct binding mode and update mode; binding to nullable values

This objective does not include: binding to a specific data source

•Implement value converters in WPF.

This objective may include but is not limited to: implementing custom value converters, implementing multivalue converters

•Implement data validation.

This objective may include but is not limited to: handling validation and providing user feedback via the error provider (WinForms) or data templates (WPF), IDataErrorInfo, validation control, form validation and control validation

•Implement and consume change notification interfaces.

This objective may include but is not limited to: implementing INotifyPropertyChanged; using INotifyCollectionChanged (ObservableCollection)

•Prepare collections of data for display.

This objective may include but is not limited to: filtering, sorting, and grouping data; LINQ; CollectionView (WPF), BindingSource object (WinForms)

•Bind to hierarchical data.

This objective may include but is not limited to: TreeView; MenuControl

•Implement data-bound controls.

This objective may include but is not limited to: using the DataGridView (WinForms) or DataGrid (WPF) control to display and update the data contained in a data source, implementing complex data binding to integrate data from multiple sources; ItemsControl-derived controls (WPF)

•Create a data template in WPF.

This objective may include but is not limited to: implementing a data template selector; using templates with ItemsControl

Enhancing the Functionality and Usability of a Solution (17%)

•Integrate WinForms and WPF within an application.

This objective may include but is not limited to: using ElementHosts within WinForms and ControlHosts within WPF; using the PropertyMap property

•Implement asynchronous processes and threading.

This objective may include but is not limited to: implementing asynchronous programming patterns; marshalling between threads; freezing UI elements; using timers; Task Parallel Library; parallel LINQ; using the dispatcher; BackgroundWorker component

•Incorporate globalization and localization features.

This objective may include but is not limited to: loading resources by locale; marking localizable elements; using culture settings in validators and converters; using language properties and rendering direction properties; working with resource files for localization; determining installed locales; regional settings

•Implement drag and drop operations within and across applications.

This objective does not include: Dynamic Data Exchange (DDE)

•Implement security features of an application.

This objective may include but is not limited to: configuring Software Restriction Policy (SRP); full trust and partially trusted security; interoperability with legacy CAS policy; User Account Control (UAC)

•Manage user and application settings.

This objective may include but is not limited to: creating application settings; creating user settings; loading and saving settings

This objective does not include: persisting to database

•Implement dependency properties.

This objective may include but is not limited to: enabling data binding and animation, property metadata, property change callbacks

Stabilizing and Releasing a Solution (17%)

•Implement a WPF test strategy.

This objective may include but is not limited to: automation peer, UI automation, IntelliTrace

•Debug XAML by using the WPF Visualizer.

This objective may include but is not limited to: accessing the Visualizer, drilling down into the visual tree, viewing and changing properties

This objective does not include: setting a breakpoint and stepping through code

•Debug WPF issues by using PresentationTraceSources.

This objective may include but is not limited to: animation, data binding, dependency properties

•Configure a ClickOnce deployment.

This objective may include but is not limited to: configuring the installation of a WinForms, WPF, or XBAP application by using ClickOnce technology; choosing appropriate settings to manage upgrades

•Create and configure a Windows Installer project.

This objective may include but is not limited to: configuring a setup project to add icons during setup, setting deployment project properties, configuring conditional installation based on operating system versions, setting appropriate Launch Conditions based on the .NET Framework version, adding custom actions to a setup project, adding error-handling code to a setup project

•Configure deployment security settings.

This objective may include but is not limited to: configuring and integrating UAC by using ClickOnce deployments; setting appropriate security permissions to deploy the application

Tópico 1 – Criando interfaces básicas

Objetos de conteúdo

- Canvas - permite a criação de layouts por coordenadas, dentro do espaço delimitado pelo Canvas
- DockPanel – Os objetos ficam alinhados um em relação ao outro orientados vertical ou horizontalmente
- Grid – Objetos organizados em linhas e colunas, funciona como uma tabela com as funções de *Rows, Cols, ColumnSpan, RowSpan* permitindo um layout simular a planilhas e HTML
- StackPanel – Os objetos são alinhados em linha horizontal ou vertical sequencialmente, e é muito útil quando os controles precisas ser redimensionados e redistribuídos na tela quando o formulário for redimensionado
- VirtualizingStackPanel – Similar ao StackPanel, mas este só mantem ativo na memória os objetos que estão visiveis
- WrapPanel – Os objetos ficam sequencialmente posicionados, permitindo quebra de linha e pode ser orientado do topo para o rodapé (ou o inverso) e da direita para esquerda (ou o inverso)

Recursos estáticos e compartilhados (dicionários)

```
<Window  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:sys="clr-namespace:System;assembly=mscorlib"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
  <Window.Resources>
    <SolidColorBrush x:Key="BrushAzul" Color="Blue" />
    <SolidColorBrush x:Key="BrushBranco" Color="White" />
   <sys:Double x:Key="Largura">100</sys:Double>
  </Window.Resources>
</Window>
```

O exemplo acima cria uma janela com dois recursos compartilhadas, a cor preenchimento e caneta Azul e Branca. A seguir, veja como é utilizado estes recursos dentro dos objetos:

```
<StackPanel HorizontalAlignment="Center">
    <TextBlock FontSize="36"
           Foreground="{StaticResource BrushAzul}"
           Text="Text" />
    <Button Background="{StaticResource BrushAzul}"
          Foreground="{StaticResource BrushBranco}"
          Height="30"
          Width="60">Text</Button>
    <Ellipse Fill="{StaticResource BrushAzul}"
           Height="{StaticResource myValue}"
           Margin="10"
           Width="{StaticResource Largura}" />
  </StackPanel>
```

Note como no botão, no circulo e no texto foram definidos a cor utilizando o recurso compartilhado criado anteriormente. Note também que utilizamos valores fixos (Largura) com o efeito de simular variáveis globais.
Para aplicar um recurso de forma programática utilize o exemplo abaixo:

```
Botao.Background = (Brush)this.TryFindResource("BrushBranco");
```

Estes recursos também podem ser compartilhados utilizando arquivos externos:

```
<Page.Resources>
  <ResourceDictionary.MergedDictionaries>
    <ResourceDictionary Source="Resources\MyResources1.xaml" />
    <ResourceDictionary Source="Resources\MyResources2.xaml" />
  </ResourceDictionary.MergedDictionaries>
</Page.Resources>
```

Os recursos compartilhados são muito úteis porque também podem ser aproveitados entre aplicações, por se compartilhar os *xaml*'s entre elas com propriedades comuns como um logo, nome da empresa, textos, etc.

Estilos

Estilos podem ser utilizados como os recursos e vinculados a um objeto. A diferença é que no recurso se utiliza o nome para dar o valor a uma propriedade, já os estilos definem tanto a propriedade quanto o valor:

```
<Page.Resources>
  <Style x:Key="MeuEstilo">
    <Setter Property="Control.Background" Value="Blue" />
    <Setter Property="Control.Foreground" Value="White" />
  </Style>
</Page.Resources>
<Label Content="Guia 70511"
    FontSize="18"
    FontWeight="Bold"
    Style="{StaticResource MeuEstilo }" />
```

No exemplo abaixo veja que o estilo é aplicado apenas a objetos *Label*:

```
<Page.Resources>
  <Style x:Key="MeuEstilo" TargetType="{x:Type Label}">
    <Setter Property="Background" Value="Blue" />
    <Setter Property="Foreground" Value="White" />
  </Style>
</Page.Resources>
```

Se acrescentamos a propriedade *x:Uid* podemos indica que o estilo é para o controle com o nome especifico, como no exemplo abaixo:

```
  <Style x:Uid="lblTexto"  x:Key="MeuEstilo" TargetType="{x:Type Label}">
```

Herdando e sobrescrevendo estilo

O código abaixo demonstra como podemos utilizer um estilo já existente e customizá-lo. Veja que o estilo *Cabeçalho* altera do *meuEstilo* o tamanho da fonte:

```
<Page.Resources>
  <Style x:Key="meuEstilo" TargetType="{x:Type Label}">
    <Setter Property="Background" Value="Blue" />
    <Setter Property="Foreground" Value="White" />
    <Setter Property="FontSize" Value="16" />
  </Style>
  <Style x:Key="Cabeçalho"
      BasedOn="{StaticResource meuEstilo}" TargetType="{x:Type Label}">
    <Setter Property="FontSize" Value="24" />
```

Animações

Um Storyboard pode ser criado para definir uma sequencia de interações que um objeto irá ter. No exemplo abaixo o objeto invisível progressivamente de 100% para 0% no intervalor de 1 segundo, de forma repetitiva e auto reversível, criando um efeito "some-e-aparece" no objeto chamado *Quadro*:

```
<Storyboard>
  <DoubleAnimation AutoReverse="True"
          Duration="0:0:1"
          From="1.0"
          RepeatBehavior="Forever"
          Storyboard.TargetName="Quadro"
          Storyboard.TargetProperty="Opacity"
          To="0.0"
</Storyboard>
```

É possível usar as *Easinc Functions* para criar efeitos profissionais como *BackEase, CircleEase, PowerEase* e outras. Por exemplo, o código abaixo utiliza o efeito *Bouncing* no quadro:

```
<Storyboard x:Name="Animacao">
  <DoubleAnimation From="30" To="200" Duration="00:00:3"
```

```
                Storyboard.TargetName="Quadro"
                Storyboard.TargetProperty="Height">
                <DoubleAnimation.EasingFunction>
                  <BounceEase Bounces="2"
                        EasingMode="EaseOut"
                        Bounciness="2" />
                </DoubleAnimation.EasingFunction>
      </DoubleAnimation>
    </Storyboard>
```

Para iniciar uma animação por código basta encontrar a animação como o código abaixo. Também é possivel utilizar *Pause, Resume, Seek, Stop*, etc.

```
var storyboard = (Storyboard)this.FindResource("Animacao");
storyboard.Begin(this, true);
```

Temas

Os temas já são conhecidos de aplicações ASP.NET e funcionam de forma similar.

1. Crie o diretório *Themes*
2. Crie o arquivo *generic.xaml*
3. Defina a seção *ResourceDictionary* com os estilos
4. Defina o diretório na aplicação host

O código abaixo seria um exemplo do arquivo genérico criando a definição do controle com as propriedades desejadas como default para um controle do tipo up-down:

```
<ResourceDictionary   xmlns:local="clr-namespace:CustomControlLibrary" ...>
 <Style TargetType="{x:Type local:NumericUpDown}">
  <Setter Property="Template">
   <Setter.Value>
    <ControlTemplate TargetType="{x:Type local:NumericUpDown}">
     <Grid Margin="3">
      <Grid.RowDefinitions>
       <RowDefinition/>
       <RowDefinition/>
      </Grid.RowDefinitions>
      <Grid.ColumnDefinitions>
       <ColumnDefinition/>
       <ColumnDefinition/>
      </Grid.ColumnDefinitions>
      <TextBlock Text="{Binding RelativeSource={x:Static
RelativeSource.TemplatedParent}, Path=Value}" />
      <RepeatButton
       Command="{x:Static local:NumericUpDown.IncreaseCommand}"
       Grid.Column="1"
       Grid.Row="0">Up</RepeatButton>
      <RepeatButton
       Command="{x:Static local:NumericUpDown.DecreaseCommand}"
       Grid.Column="1"
       Grid.Row="1">Down</RepeatButton>
     </Grid>
    </ControlTemplate>
   </Setter.Value>
  </Setter>
 </Style>
</ResourceDictionary>
```

Para utilizar um tema para toda a aplicação coloque no arquivo *AssemblyInfo.cs* o código abaixo:

```
[assembly: ThemeInfo(
  ResourceDictionaryLocation.None,
```

ResourceDictionaryLocation.SourceAssembly)]

Também é possível indicar em cada janela ou controle um tema, mas neste caso o código acima deve ser alterado para *ResourceDictionaryLocation.SourceAssembly* indicando que cada página ou janela terá o seu.

Trigger de animação

No exemplo abaixo, note que utilizamos o evento *MouseEnter* para indicar que ao passar o mouse a animação começa com o objeto ficando opaco e ao tirar o mouse (*MouseLeave)* o objeto reaparece:

```
<Rectangle.Triggers>
 <EventTrigger RoutedEvent="Mouse.MouseEnter">
  <EventTrigger.Actions>
   <BeginStoryboard>
    <Storyboard>
     <DoubleAnimation Duration="0:0:1"
              Storyboard.TargetProperty="Opacity"
              To="0.0" />
    </Storyboard>
   </BeingStoryboard>
   </EventTrigger.Actions>
 </EventTrigger>
 <EventTrigger RoutedEvent="Mouse.MouseLeave">
  <EventTrigger.Actions>
   <BeginStoryboard>
     <DoubleAnimation Duration="0:0:1"
              Storyboard.TargetProperty="Opacity" />
   </BeginStoryboard>
  </EventTrigger.Actions>
 </EventTrigger>
</Rectangle.Triggers>
```

O exemplo abaixo, mais completo demonstra como fazer a animação em um quadro, com controle de inicio, pausa, resumir, etc:

```
<Page>
 <StackPanel Margin="20" >
  <Rectangle Name="myRectangle"
   Width="100" Height="20" Margin="12,0,0,5" Fill="#AA3333FF" HorizontalAlignment="Left" />
  <StackPanel Orientation="Horizontal" Margin="0,30,0,0">
   <Button Name="BeginButton">Begin</Button>
   <Button Name="PauseButton">Pause</Button>
   <Button Name="ResumeButton">Resume</Button>
   <Button Name="SeekButton">Seek</Button>
   <Button Name="SkipToFillButton">Skip To Fill</Button>
   <Button Name="SetSpeedRatioButton">Triple Speed</Button>
   <Button Name="StopButton">Stop</Button>
   <StackPanel.Triggers>
    <EventTrigger RoutedEvent="Button.Click" SourceName="BeginButton">
     <BeginStoryboard Name="MyBeginStoryboard">
      <Storyboard >
       <DoubleAnimation
        Storyboard.TargetName="myRectangle"
        Storyboard.TargetProperty="Width"
        Duration="0:0:5" From="100" To="500" />
      </Storyboard>
     </BeginStoryboard>
    </EventTrigger>
    <EventTrigger RoutedEvent="Button.Click" SourceName="PauseButton">
     <PauseStoryboard BeginStoryboardName="MyBeginStoryboard" />
```

```
        </EventTrigger>
        <EventTrigger RoutedEvent="Button.Click" SourceName="ResumeButton">
          <ResumeStoryboard BeginStoryboardName="MyBeginStoryboard" />
        </EventTrigger>
         <EventTrigger RoutedEvent="Button.Click" SourceName="SeekButton">
           <SeekStoryboard
            BeginStoryboardName="MyBeginStoryboard"
            Offset="0:0:1" Origin="BeginTime" />
        </EventTrigger>
        <EventTrigger RoutedEvent="Button.Click" SourceName="SkipToFillButton">
          <SkipStoryboardToFill BeginStoryboardName="MyBeginStoryboard" />
        </EventTrigger>
        <EventTrigger RoutedEvent="Button.Click" SourceName="StopButton">
          <StopStoryboard BeginStoryboardName="MyBeginStoryboard" />
        </EventTrigger>
        <EventTrigger RoutedEvent="Button.Click" SourceName="SetSpeedRatioButton">
          <SetStoryboardSpeedRatio SpeedRatio="3" BeginStoryboardName="MyBeginStoryboard" />
        </EventTrigger>
      </StackPanel.Triggers>
    </StackPanel>
  </StackPanel>
</Page>
```

Tópico 2 - Criando Interface de usuários com técnicas avançadas

Trabalhando com eventos

No exemplo do tópico precedente vemos os *EventTriggers* que indicam um método para o eventos de pausar, rodar e parar um vídeo.
Porem, existem os eventos mapeados no container. Veja neste exemplo como é feito para criar um evento roteado:

```
<StackPanel Button.Click="OnButtonClick">
  <Button>Sim</Button>
  <Button>Não</Button>
</StackPanel>
```

Este tipo de evento é roteado entre os controles que estão dentro do container *Stack* e o código para controla-los seria utilizando o nome ou outra propriedade do botão:

```
private void OnButtonClick(object sender, RoutedEventArgs e)
{
        Button obj = (Button)sender;
        If(obj.Text=”Yes”) ….
}
```

Um evento pode ser vinculado de forma programática como o exemplo abaixo nos casos de um *user control* para indicar o evento daquele objeto:

```
AddHandler(TextBox.TextChangedEvent, new RoutedEventHandler(Audit_TextChanged), true);
```

Os eventos ocorrem em pares *Tunneling* e *Bubble*, sendo que o primeiro é executado para um evento do objeto container para os objetos filhos e o outro dos filhos para os containers.
Os eventos tunneling ocorrem antes dos eventos bubble são precedidos da palavra *Preview*, como por exemplo, *PreviewMouseRightButtonDown*. Eles são úteis para quando se deseja fazer alguma tarefa ANTES do processo do evento, por exemplo, manipular alguma variável indicando que o click está sendo processado.
Você pode cancelar a execução da cascata de eventos utilizando a instrução *e.Handled = false*.
Uma referencia a todos os eventos é http://msdn.microsoft.com/en-us/library/system.windows.uielement_events.aspx

Associação com comandos

O WPF contem muitos comandos de UI, operações e outros que utilizamos na programação. Um exemplo são os comandos que manipulam a área de troca (Copy and Past). São chamados de comandos pré-definidos.
Estes comandos podem ser mapeados a objetos WPF como a sintaxe a seguir onde um item de menu foi vinculado a uma ação padrão de janelas (*Close*), executando na sequencia o código em C#, que não está descrito mas teria o nome *OnWindowClose*:

```
<Window x:Class="Commands.MainWindow" ...>
   <Window.CommandBindings>
     <CommandBinding   Command="Close"  Executed="OnWindowClose" />
  </Window.CommandBindings>
  <DockPanel>
    <Menu DockPanel.Dock="Top">
      <MenuItem Header="_File">
        <MenuItem Header="_Close" Command="Close" />
      </MenuItem>
    </Menu>
    <Grid />
  </DockPanel>
 </Window>

public partial class MainWindow : Window
{
        …
```

```
private void OnWindowClose(object sender, ExecutedRoutedEventArgs e)
{
   this.Close();
}
        ...
```

Outros exemplos seria o ApplicationCommands.Cut, ApplicationCommands.Copy, ApplicationCommands.Past, etc. Este commandos estão presentes nas classes ApplicationCommands, NavigationCommands, MediaCommands, EditingCommands e ComponentCommands.

Os CommandBindings podem ter *Executed* e *CanExecute*, como o exemplo abaixo:

```
<Window.CommandBindings>
  <CommandBinding Command="ApplicationCommands.Open"
          Executed="OpenCmdExecuted"
          CanExecute="OpenCmdCanExecute"/>
</Window.CommandBindings>
```

Veja mais exemplos e referencias adicionais em http://msdn.microsoft.com/en-us/library/ms604577.aspx e procure pelas classes citadas acima.

Modificando a interface em runtime

É comum em aplicações termos a necessidade de criar controles em tempo de execução mas em WPF o método é um pouco diferenciado. No exemplo abaixo utilizamos uma janela com um Canvas, exatamente para demonstrar como se faz tanto a adição de um novo controle, como o posicionamento fixo:

```
private void button1_Click(object sender, RoutedEventArgs e)
{
   Button Botao = new Button();
   Botao.Content = "Novo Botão";
   Botao.Width = 200;
   Botao.FontSize = 24;
   Canvas.SetLeft(Botao, 10);
   Canvas.SetTop(Botao, 100);
   canvas1.Children.Add(Botao);
}
```

Note que o posicionamento não é feito no controle, já que existem os 6 diferentes tipos de containers, cada um com um tipo diferente de posicionamento. Por exemplo, se estivéssemos utilizando o StackPanel apenas acrescentaríamos o controle e ele ficaria na sequencia dos que já existem, se fosse o Grid passaríamos a linha e coluna e assim por diante.

Para aplicar um template ou style programaticamente utilize o exemplo abaixo onde foi encontrado o estilo *LabeledListBox* e depois aplicado a uma nova lista, onde tanto o template quanto o estilo foram definidos.

```
ListBox newListBox = new ListBox();
Style mystyle = (Style)FindResource("LabeledListBox");
newListBox.Style = (Style)FindResource("LabeledListBox");
newListBox.ApplyTemplate();
```

Templates

Diferente de um estilo, o *Template* cria o padrão de um tipo de objeto. Com ele podemos indicar as propriedades de um objeto ao cria-lo, como no exemplo abaixo para botões:

```
<Style TargetType="Button">
  <Setter Property="Template">
   <Setter.Value>
    <ControlTemplate TargetType="Button">
     <Grid>
      <Ellipse Fill="LightBlue" />
      <ContentPresenter HorizontalAlignment="Center"
                VerticalAlignment="Center" />
     </Grid>
    </ControlTemplate>
   </Setter.Value>
```

```
 </Setter>
</Style>
```

Controles customizados (UserControls)

Assim como já existia em versões anteriores do Visual Studio é possível criar controles customizados para depois integrá-los como componentes na sua aplicação.
A criação é simples, com os elementos de página ou janela como o exemplo a seguir que tem um botão, bloco de texto e evento ao clicar o botão:

```
<UserControl x:Class="WpfApplication1.UserControl1" …>
  <Grid>
    <Button Content="Button" Height="23" HorizontalAlignment="Left" Margin="100,12,0,0"
Name="button1" VerticalAlignment="Top" Width="75" Click="button1_Click" />
    <TextBlock Height="23" HorizontalAlignment="Left" Margin="48,43,0,0" Name="textBlock1"
Text="TextBlock" VerticalAlignment="Top" Width="190" FontStretch="Normal" TextAlignment="Center"
/>
  </Grid>
</UserControl>
```

Os controles customizados são muito úteis para reaproveitamento de objetos visuais, podendo ser utilizados para montar validações, controles de navegação e diversos outros objetos que são iguais, alterando propriedades especificas para se adequar a um ou outro formulário ou janela.
Para criar estas propriedades customizadas você pode utilizar variáveis públicas ou então propriedades. No exemplo abaixo temos uma propriedades para alterar o texto que está no botão:

```
public partial class UserControl1 : UserControl
{
  public string TextoBotao
  {
    get { return button1.Text; }
    set { button1.Text = value; }
  }
```

Para alterar este valor ao utilizar em uma janela veja o exemplo a seguir:

```
<my:UserControl1 x:Name="userControl11" TextoBotao="Teste" Canvas.Left="20" Canvas.Top="62" />
```

Ao utilizar este controle em uma aplicação é possível vincular uma propriedades a um método, o que permitirá customizar o comportamento quando a propriedade for alterada, como o exemplo abaixo:

```
UserControl11.PropertyMap.Remove("TextoBotao");
UserControl11.PropertyMap.Add("TextoBotao", new PropertyTranslator(OnTextoAlterado));
```

Criando objetos gráficos

O WPF possui basicamente dois objetos para trabalhar com gráficos, que suprem todas as necessidades: Rectangle e Ellipse. Veja no exemplo abaixo um ciculo, quadrado e linha utilizando estes objetos:

```
<Ellipse Canvas.Left="12" Canvas.Top="134" Height="92" Name="Circulo" Stroke="Black" Width="93" />
<Rectangle Canvas.Left="161" Canvas.Top="148" Height="41" Name="Quadro" Stroke="Black" Width="67"
/>
<Rectangle Canvas.Left="58" Canvas.Top="246" Height="1" Name="Linha" Stroke="Black" Width="173" />
```

Para inserir um objeto gráfico manualmente em uma janela pode ser utilizado o evento qualquer evento, diferentemente das aplicações Windows Forms anteriores onde apenas o evento *OnPaint* permitia este interação. Note o código abaixo que acrescenta um circulo ao clicar em um botão:

```
Ellipse Circulo2 = new Ellipse();
Circulo2.Height = 30; Circulo2.Width = 60;
Circulo2.Stroke = Brushes.Black;
canvas1.Children.Add(Circulo2);
```

Um importante objeto que é utilizado em todos os objetos gráficos é o *Brush* para se criar uma cor de caneta ou então o *Brushes* quando se deseja utilizar um enumerador com as cores padrões.
Uma interessante mudança é como são tratados objetos gráficos. Nas aplicações Windows Forms é responsabilidade da aplicação redesenhar ou manter gráficos, já que não são vetorias. Já aplicações WPF os gráficos são vetoriais e mantidos pelo sistema, ou seja, não é o WPF que se responsabilidade pela manutenção visual dos gráficos. Mais detalhes em http://msdn.microsoft.com/en-us/library/ms748373.aspx

Conteúdo multimídia

As aplicações WPF podem lidar com duas formas de visualização de conteúdo multimídia, um é o objeto *MediaElement* que apresenta o conteúdo na página e o outro é o *MediaPlayer* que utiliza-se para manipulação manual e programática. O primeiro é ideal para mostrar vídeos ou permitir interação com musicas, enquanto o segundo é ideal para musicas de fundo, por exemplo.

O exemplo abaixo mostra um filme passando com os controles já pré-definidos para um tocador de media:

```
<MediaElement Height="120" Name="ME1" Width="160" Source="a.wmv" LoadedBehavior="Play" />
```

Este objeto é muito fácil de ser utilizado e tem diversos eventos disponiveis como *MediaOpened*, *MediaFailed, MediaEnd* além dos comuns a todos os controles. Também conta com propriedades simples e intuitavas como *CanPlay* e *CanPause* e os métodos óbvios *Play, Pause, Resume* e *Stop*.

Para fazer um vídeo não executar automaticamente, altere a propriedade *LoadedBehavior* para *none*.

Outro objeto é o *MediaPlayer* que não é um controle visual, mas é colocado dentro de outros objetos, como o exemplo a seguir:

```
<Border>
  <Border.Background>
    <DrawingBrush>
      <DrawingBrush.Drawing>
        <VideoDrawing x:Name="videoSurface"
                Rect="0,0,300,200" />
      </DrawingBrush.Drawing>
    </DrawingBrush>
  </Border.Background>
</Border>
```

A sua manipulação é feita por código, como o exemplo a seguir que define o video

```
MediaPlayer player = new MediaPlayer();
player.Open(new Uri(@"a.wmv"));
this.videoSurface.Player = player;
player.Play();
```

Codificando e decodificando imagens (conversão)

Uma funcionalidade importantíssima para aplicações hoje é fazer upload de imagens que podem estar em vários formatos. Para isso contamos com funções que decodificam uma imagem e podem transformá-la, ou codifica-la novamente em outro formato.

Este codificadores são *BmpBitmapDecoder, GifBitmapDecoder, JpegBitmapDecoder* e *PngBitmapDecoder* (todos tem também o "Encoder").

Note que no exemplo abaixo o usuário sobe um JPEG e a imagem é convertida em PNG;

```
JpegBitmapDecoder decoder =
  new JpegBitmapDecoder(new Uri("Foto.jpg"),
    BitmapCreateOptions.PreservePixelFormat,
    BitmapCacheOption.None);

//Converte em PNG
using (Stream outputStream = new FileStream
     ("Foto.png", FileMode.Create))
{
  PngBitmapEncoder encoder = new PngBitmapEncoder();
  encoder.Frames.Add(decoder.Frames[0]);
  encoder.Save(outputStream);
}
```

Utilizando imagens

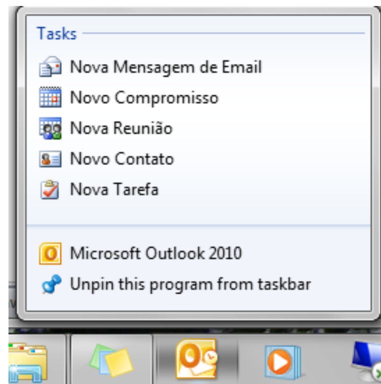
Também é importante ressaltar que tres objetos podem ser criados para trabalhar com imagens:

1. Image – É o objeto básico, mostra uma imgem
2. ImageBrush – Utiliza uma imagem para servir de caneta

3. ImageDrawing – Similar ao Imagem, porem tem métodos mais complexos, como transformações da imagem, como por exemplo, sépia, BW, etc.

Menus

Existem vários tipos de menu possíveis em aplicações com WPF e WinForms e 4.0
A mais comum é a classe de menus e a de contexto. Os menus de contexto são uteis para indicar funções especificas para um determinado controle ou item da tela do usuário.
Um menu de contexto é criado utilizando o objeto a partir da *toolbox* e para liga-lo a um controle basta utilizar a propriedade *ContextMenu* do objeto desejado.
Outro tipo de menu novo é o *JumpList* que é especifica para o Windows 7 e mostra as opções desejadas, como a imagem a seguir do Outlook:



Para criar estes itens utilize o código abaixo como exemplo na sua aplicação:

```
JumpList jumpList = new JumpList();
JumpList.SetJumpList(Application.Current, jumpList);
JumpTask jumpTask = new JumpTask();
jumpTask.Title = jumpTask.Description = namesToUse[0];
jumpTask.CustomCategory = "Funções";
jumpTask.ApplicationPath = "Cadastro de Clientes;
jumpList.JumpItems.Add(jumpTask);
jumpList.Apply();
```

Este exemplo cria um item sobre uma categoria *Funções* com a descrição *Cadastro de Clientes*. É claro que é necessário após o objeto criado fazer o *handler* para o evento *click* do *JumpTask*.

Tópico 3 - Trabalhando com dados (Data bindings)

Implementando data bindings

Fazer a implementação de dados em aplicações Windows Forms e WPF é muito diferente.
As aplicações Windows Forms são baseadas em DataSet, TableAdapters e podem ser facilmente manipuladas pelos controles BindingNavigator, BindingSource, etc.
Já as aplicações WPF trabalham de forma diferente, podendo ter múltiplos tipos de binding, por exemplo, binding a resources, entre controles, etc.
Como o maior foco do exame 70-511 são as aplicações baseadas em WPF, daremos ênfase a este modelo.

Binding com propriedades

Este é um interessante exemplo de binding para WPF. Note a classe criada abaixo:

```
namespace WpfApplication1
{
   public class DadosExemplo
   {
      private string Nome1;
      private int Idade1;

      public string Nome
      {
         get { return Nome1; }
         set  { Nome1 = value;  }
      }
      public int Idade
      {
         get { return Idade1; }
         set  { Idade1 = value;    }
      }

      public DadosExemplo()
      {
         Nome1 = "Marcelo"; Idade1 = 39;
      }
   }
}
```

Esta classe contem dados simples e pertence ao mesmo namespace que a janela a seguir. Note que o bind foi feito para uma propriedade especifica da classe:

```
<DockPanel xmlns:c="clr-namespace:WpfApplication1">
   <DockPanel.Resources>
      <c:DadosExemplo x:Key="DadosExemplo"/>
   </DockPanel.Resources>
   <Button Content="{Binding Path=Nome,Source={StaticResource DadosExemplo}}" Height="30"
Width="150" Fallback=”Indefinido”/>
</DockPanel>
```

Também seria possível aplicar as propriedades usando os elementos do objeto botão:

```
<Button>
   <Button.Content>
      <Binding Path="Nome" Source="{StaticResource DadosExemplo}" />
   </Button.Content>
</Button>
```

Pode-se definir um contexto padrão para os dados e vincular diretamente a propriedade:

```
<DockPanel.Resources>
   <c:DadosExemplo x:Key="DadosExemplo"/>
```

```
</DockPanel.Resources>
<DockPanel.DataContext>
  <Binding Source="{StaticResource DadosExemplo}" />
</DockPanel.DataContext>
<Button Content="{Binding Path=Nome}" />
<Label Content="{Binding Path=Idade}" />
```

Note que em um dos *bindings* temos a instrução *FallbackValue* que permite colocar um valor fixo em caso de problema ao executa a leitura dos dados.

Binding com XML

O bind com XML também é relativamente simples, bastando definir o XML ou sua localização e o XPath para a pesquisa dos dados.

Note que o exemplo abaixo contem o XML embutido e os seus dados:

```
<StackPanel>
  <StackPanel.Resources>
    <XmlDataProvider x:Key="inventoryData" XPath="Inventory/Books">
      <x:XData>
        <Inventory xmlns="">
          <Books>
            <Book ISBN="0-7356-0562-9" Stock="in" Number="9">
              <Title>XML in Action</Title>
              <Summary>XML Web Technology</Summary>
            </Book>
            <Book ISBN="0-7356-1288-9" Stock="out" Number="7">
              <Title>Inside C#</Title>
              <Summary>C# Language Programming</Summary>
            </Book>
            <Book ISBN="0-7356-1448-2" Stock="out" Number="4">
              <Title>Microsoft C# Language Specifications</Title>
              <Summary>The C# language definition</Summary>
            </Book>
          </Books>
        </Inventory>
      </x:XData>
    </XmlDataProvider>
  </StackPanel.Resources>

  <ListBox>
    <ListBox.ItemsSource>
      <Binding Source="{StaticResource inventoryData}"
XPath="*[@Stock='out'] | *[@Number>=8 or @Number=3]"/>
    </ListBox.ItemsSource>

    <ListBox.ItemTemplate>
      <DataTemplate>
        <TextBlock Text="{Binding XPath=Title}">
          <TextBlock.Tooltip>
            <TextBlock Text="{Binding Path=Attributes[0].Value}" />
          </TextxBlock.Tooltip>
        </TextBlock>
      </DataTemplate>
    </ListBox.ItemTemplate>
  </ListBox>
</StackPanel>
```

Binding com outros controles

Uma interessante possibilidade é criar o binding de um controle a outro.
No exemplo abaixo a cor de fundo do Canvas é determinado pela cor escolhida na combo:

```
<StackPanel>
 <ComboBox x:Name="Cores" SelectedIndex="0">
  <ComboBoxItem>Green</ComboBoxItem>
  <ComboBoxItem>LightBlue</ComboBoxItem>
  <ComboBoxItem>Red</ComboBoxItem>
 </ComboBox>
 <Canvas Background="{Binding ElementName=Cores,
                Path=SelectedItem.Content}"
     Height="100"
     Width="100" />
</StackPanel>
```

Notificações de alteração para objetos binding

Uma importante interface a ser implementada nos objetos criados é a *INotifyPropertyChanged* que permite notificar os objetos utilizando a fonte de que houve alterações.
Note o exemplo da classe anterior, agora com a interface para notificação, com o evento *PropertyChanged* que é utilizado para avisar o host de que houve alterações nos controles utilizados:

```
namespace WpfApplication1
{
  public class DadosExemplo : INotifyPropertyChanged
  {
    private string Nome1;
    private int Idade1;

    public string Nome
    {
      get { return Nome1; }
      set
      {
        Nome1 = value;
        PropertyChanged(this, new PropertyChangedEventArgs("Nome"));
      }
    }
    public int Idade
    {
      get { return Idade1; }
      set
      {
        Idade1 = value;
        PropertyChanged(this, new PropertyChangedEventArgs("Idade"));
      }
    }

    public event PropertyChangedEventHandler PropertyChanged;

    public DadosExemplo()
    {
      Nome1 = "Marcelo"; Idade1 = 39;
    }
  }
}
```

Binding com banco de dados

A classe a seguir retorna um dataset a partir de um banco de dados SQL Server para um objeto ListBox:

```
DataSet meuDataSet = new DataSet();

private void OnInit(object sender, EventArgs e)
{
    SqlConnection conn = new SqlConnection(connString);
    SqlDataAdapter adapter =  new SqlDataAdapter("SELECT * FROM X;", conn);
    adapter.Fill(meuDataSet, "X");
    meuListBox.DataContext = meuDataSet;
}
```

O código que implementaria o ListBox seria:

```
<ListBox Name="myListBox" Height="200"  ItemsSource="{Binding Path=X}" />
```

Binding com LINQ

O Linq é muito útil para rápidos filtros sobre objetos em memória. Imagine que você tenha um XML com uma lista de dados de planetas e lhe interesse mostrar em um listbox os planetas.

Você pode implementar o código a seguir:

```
<Window.Resources>
  <ObjectDataProvider x:Key="planets"
            MethodName="Load"
            ObjectType="{x:Type xlinq:XElement}">
    <ObjectDataProvider.MethodParameters>
      <sys:String>SolarSystemPlanets.xml</sys:String>
    </ObjectDataProvider.MethodParameters>
  </ObjectDataProvider>
  ...
</Window.Resources>
...
<Grid DataContext="{Binding Source={StaticResource planets},
Path=Elements}">
  <ListBox ItemsSource="{Binding}" ... />
  ...
</Grid>
```

Note que o método *xlinq* faz a pesquisa dos elementos baseados nos planetas e traz todos os *Elements*, mas poderia trazer apenas um, por exemplo utilizando E*lement["Sistema Solar"].Value*

Data Template

Alguns objetos como ListBox podem mostrar dados múltiplos, com diversas colunas.

O exemplo abaixo mostra a criação de um conjunto de três TextBlocks que mostram os dados do cliente, porem note que estão em um mesmo DataTemplate

```
<Window.Resources>
  ...
  <DataTemplate x:Key="meuDataTemplate">
    <StackPanel>
      <WrapPanel>
        <TextBlock Text="{Binding Path=Nome}" />
        <TextBlock Text="{Binding Path=Sobrenome}" />
      </WrapPanel>
      <TextBlock Text="{Binding Path=Idade}" />
    </StackPanel>
  </DataTemplate>
</Window.Resources>
<ListBox
     ItemsSource="{Binding Source={StaticResource list}}"
```

```
ItemTemplate="{StaticResource meuDataTemplate}" />
```

Note que o ListBox mostrará os tres TextBlocks como um conjunto. Isso é possível em aplicação WinForms apenas clicando sobre um ListBox com o botão direito e escolhendo *ItemTemplate, HeaderTemplate* e outros.

Conversão de dados para binding

Muitas vezes temos a necessidade de trabalhar diretamente com conversões, por exemplo, com tipo de dados DateTime. Nestes casos utilizamos as classes de conversão que implementam a interface *IvalueConverter*, como o exemplo a seguir:

```
 [ValueConversion(typeof(DateTime), typeof(String))]
public class DateConverter : IValueConverter
{
  public object Convert(object value, Type targetType,
    object parameter, CultureInfo culture)
  {
    DateTime date = (DateTime)value;
    return date.ToShortDateString();
  }

  public object ConvertBack(object value, Type targetType,
    object parameter, CultureInfo culture)
  {
    string strValue = value.ToString();
    DateTime resultDateTime;
    if (DateTime.TryParse(strValue, out resultDateTime))
    {
      return resultDateTime;
    }
    return value;
  }
}
```

Note a diretiva indicando que a conversão é feita de string para datetime e vice-versa. Este modelo serve para qualquer outra conversão que seja necessária.
O exemplo abaixo mostra o uso da conversão:

```
<Page.Resources>
  <src:DateConverter x:Key="dateConverter" />
</Page.Resources>
...
<TextBlock Text="{Binding Path=StartDate,  Converter={StaticResource dateConverter}}" />
```

Como pode ser visto no exemplo, a fonte de dados não está definida, mas poderia ser acrescentada já que a função de conversão é independente da propriedade *Source*.

BindingSource para WinForms

No caso de aplicações WinForms é possível utilizar controles que fazem a navegação sincronizada, similar ao que vimos até o momento com aplicações WPF, o que é muito útil para aplicações do tipo lista principal e lista de detalhes (*Master-Details*).
O primeiro passo é criar e popular um *Dataset* e criar o objeto *BindingSource*, como o exemplo a seguir. Os objetos *Dataset* e *DataAdapter* tem tipos definidos pois foram criados a partir de um *DataSet* da aplicação:

```
//Cria os objetos básicos
private ClientesDataSet clientesDataSet;
private System.Windows.Forms.BindingSource pFBindingSource;
private ClientesDataSetTableAdapters.PFTableAdapter pFTableAdapter;
//Código que popula o dataset omitido propositalmente
pFBindingSource.DataMember = "PF";
pFBindingSource.DataSource = this.clientesDataSet;
this.dataGridView1.DataSource = this.pFBindingSource;
this.textBox1.DataBindings.Add(new Binding("Text", this.pFBindingSource, "CPF", true));
```

Note que tanto o objeto *DataGrid* quanto o objeto *TextBox* estão ligados ao mesmo *BindingSource*, sincronizando a navegação e operações.

Mostrando ícones de validação

No caso de aplicações WinForms basta utilizar o controle *ErrorProvider* e definir sua propriedade *SetError* como a seguir:

```
ErrorProvider1.SetError(TextBox1, “Valor incorreto”)
```

O objeto *ErrorProvider* também pode ser vinculado diretamente a um *datasource*, com a instrução:

```
errorProvider1.DataSource = "ABC";
```

Mas o processo de mostrar um indicativo de aviso no WPF é bem mais complexo. Note primeiro a classe abaixo que cria um template vinculado a um trigger para o caso de o *HasError* estiver verdadeiro. Neste caso é utilizado o *ToolTip* para mostrar o primeiro erro da coleção:

```
<Style x:Key="textBoxStyle" TargetType="{x:Type TextBox}">
  <Style.Triggers>
    <Trigger Property="Validation.HasError" Value="true">
      <Setter Property="ToolTip"
        Value="{Binding RelativeSource={RelativeSource Self},
               Path=(Validation.Errors)[0].ErrorContent}"/>
    </Trigger>
  </Style.Triggers>
</Style>
```

O passo seguinte é vincular o controle a ser validado ao template acima:

```
<TextBox Style="{DynamicResource textBoxStyle}"  Validation.ErrorTemplate="{DynamicResource
errorTemplate}">
  <TextBox.Text>
    <Binding Path="StartDate" UpdateSourceTrigger="PropertyChanged">
      <Binding.ValidationRules>
        <FuturedateRule />
      </Binding.ValudationRules>
    </Binding>
  </TextBox.Text>
</TextBox>
```

Como pode ser visto neste caso, se ocorrer um erro na validação feita no evento *Validate* da classe *FutureDateRule*, aparecerá uma mensagem com o ToolTip que retornará a propriedade informada no evento como mensagem.

Um exemplo de classe que retornaria o erro para o código acima seria:

```
public class FutureDateRule : ValidationRule
{
  public override ValidationResult Validate(object value,  CultureInfo cultureInfo)
  {
    DateTime date;
    try
    {    date = DateTime.Parse(value.ToString());       }
    catch (FormatException)
    {   return new ValidationResult(false,    "Data inválida ao converter.");      }
    if (DateTime.Now.Date > date)
          return new ValidationResult(false,    "Data deve ser maior do que hoje.");
    else
           return ValidationResult.ValidResult;
  }
}
```

Note o uso da herança com a classe *ValidationRule* e o *override* no método que faz a validação.

Erros em regras de negócio

No caso acima analisamos como fazer um feedback gráfico para o erro de negócio, mas e se quisermos validar uma classe de dados utilizando propriedades desta?

Neste caso utilizamos a implementação da interface IDataErrorInfo, como o exemplo anteriormente utilizado:

```
namespace WpfApplication1
{
  public class DadosExemplo : iDataErrorInfo
  {
    private string Nome1;
    private int Idade1;

    public string Nome
    {
      get { return Nome1; }
      set  { Nome1 = value;  }
    }
    public int Idade
    {
      get { return Idade1; }
      set  { Idade1 = value;    }
    }

    public DadosExemplo()
    {
      Nome1 = "Marcelo"; Idade1 = 39;
    }
   public string Error
  {
     get {  return null;  }
  }
  public string this[string name]
  {
    get
    {
      if (name == "Idade")   //Propriedade a ser validada
      {
        if (this.Idade < 0 || this.Idade> 100)
          return “Idade inválida.";
      }
      return null;
 }
}
```

Note que não é criado um método e sim uma propriedade como um enumerador C*lasse[Propriedade]* e este enumerador faria o retorno da string com o erro ou nulo se correto.

Neste caso a implementação pareceria muito com as anteriores, mas com o método indicado *DataErrorValidationRule* que é o padrão a ser utilizado nestes casos:

```
<TextBox DataContext="{StaticResource DadosExemplo}">
  <TextBox.Text>
    <Binding>
      <Binding.ValidationRules>
        <DataErrorValidationRule />
      </Binding.ValidationRules>
    </Binding>
  </TextBox.Text>
</TextBox>
```

Tratamento de erros não gerenciados na Aplicação

Acima vimos um exemplo de tratamento de erro para validação.

No exemplo abaixo vemos uma classe para tratamento de erros que escaparem dos blocos de *try...catch* e servem como tratamento genérico para toda a aplicação. O motivo é a herança e o fato de estarmos lidando diretamente com a classe principal (*App*):

```
public partial class App : Application
{
   public App()
   {
      this.DispatcherUnhandledException +=   new DispatcherUnhandledExceptionEventHandler(
             OnDispatcherUnhandledException);
   }

   private void OnDispatcherUnhandledException(object sender, DispatcherUnhandledExceptionEventArgs
e)
   {
     //Código a ser executado
   }
}
```

Tópico 4 – Usabilidade e funcionalidades avançadas

Integrando controles WPF com controles WinForm

Esta integração não é complexa, basta utilizar o controle *WindowsFormsHost* como no exemplo abaixo e mapear as propriedades desejadas. Isso pode acontecer porque alguns controles WinForms não existem no WPF, e o exemplo do MaskedEdit é comum:

```
<Window x:Class="WpfApplication1.Window1"
 xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
 xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
 xmlns:wf="clr-
namespace:System.Windows.Forms;assembly=System.Windows.Forms">
  <Canvas>
    <WindowsFormsHost>
      <wf:MaskedTextBox x:Name="mtb" Mask="00/00/0000" />
    </WindowsFormsHost>
  </Canvas>
</Window>
```

Note que nas declarações do objeto *Window* foi referenciado o namespace de WinForms.
Para mapear as propriedades do MasketEdit com métodos do WPF é necessário utilizar o PropertyMap, como o exemplo a seguir:

```
private void AddMarginMapping()
{
  elemHost.PropertyMap.Add(
    "Margin",
    new PropertyTranslator(OnMarginChange));
}
private void OnMarginChange(object h, String propertyName, object value)
{
  ElementHost host = h as ElementHost;
  Padding p = (Padding)value;
  System.Windows.Controls.MaskedEdit wpfMask =
    host.Child as System.Windows.Controls.MaskedEdit;
  Thickness t = new Thickness(p.Left, p.Top, p.Right, p.Bottom );
  wpfMask.Margin = t;
}
```

Porque foi necessário fazer isso?   Porque como estamos executando o WPF, é necessário que se o programador do tentar alterar o tamanho do formulário ou do controle não acontecerá nada nas propriedades do *MaskedEdit*, no caso a margem. Ou seja, alterou a propriedade margem do WPF o evento *OnMarginChange* acontecerá e ajustamos o *MasketEdit*.
Ler dados dos controles já é mais simples, como o exemplo abaixo:

```
(this.wfs.Child MaskedEdit).Date;
```

Integrando controles WinForms com WPF

O inverso é mais simples pois temos um controle gráfico para fazer isso chamado de *ElementHost*.
Porem, note que diferente do exemplo acima não podemos usar controles apenas, e sim *userControls* criados em WPF. O exemplo abaixo mostra os códigos gerados pela IDE:

```
this.elementHost1 = new System.Windows.Forms.Integration.ElementHost();
this.userControl11 = new WpfApplication1.UserControl1();
this.elementHost1.Child = this.userControl11;
```

Como podemos ver, o código é simples e automatizado.
É possível usar o *PropertyMap* como no exemplo acima, mas as propriedades podem ser acessadas diretamente utilizando o userControl11 definido na criação pelo IDE:

```
userControl11.TextoBotao = "XYZ";
```

Processos assíncronos – Dispatcher

Um processo assíncrono por meio do *Dispatcher* é o meio programático de trabalhar com múltiplos processamentos. Uma de suas características é que pode manipular os controles da janela.
O exemplo a seguir cria um processo assíncrono vinculado ao botão:

```
private void Button_Click(object sender, RoutedEventArgs e)
{
    Botao.Dispatcher.BeginInvoke(new Action(Teste),
System.Windows.Threading.DispatcherPriority.Background);
}
private void Teste()
{ Console.WriteLine("Executado...");  }
```

Os diferentes tipos, ou prioridades de processamento são definidos pelo enumerador *DispatcherPriority*, veja todos em http://msdn.microsoft.com/en-us/library/system.windows.threading.dispatcherpriority.aspx. Neste caso utilizamos o *BackGround* indicando que o processo será todo feito com baixa prioridade no sistema. O evento que será executado é o *Teste*().
Apesar da *thread* poder alterar os controles gráficos isto precisa ser feito de forma segura, para isso é possível testar se o controle permitirá a modificação por *thread* com o código a seguir:

```
private void Teste()
{
  if (!Texto.Dispatcher.CheckAccess())
    Botao.Dispatcher.BeginInvoke(new Action(Teste),
System.Windows.Threading.DispatcherPriority.Background);
  else
  {
    Botao.Content = "Alterado...";
    Texto.Content = "Alterado...";
  }
}
```

Note que se o controle *Texto* não puder ser acessado naquele momento executamos a *thread* novamente até conseguirmos.
Alem do método acima baseado em interação direta do usuário, também é possível criar *timers* manuais, como se cria para aplicações WinForms com *timers* gráficos:

```
private void Button_Click(object sender, RoutedEventArgs e)
{
  DispatcherTimer MeuTimer = new DispatcherTimer(DispatcherPriority.Background, Botao.Dispatcher);
  MeuTimer.Interval = TimeSpan.FromSeconds(2);
  MeuTimer.Tick += new EventHandler(MeuTimer_Tick);
}

void MeuTimer_Tick(object sender, EventArgs e)
{
  throw new NotImplementedException();
}
```

Este exemplo criou o *timer* e o *dispatcher* sobre o objeto Botão e a cada 2 segundos o evento *Tick* é executado.

Processos assíncronos – ThreadPool

Assim como *Dispatcher* é programático, mas não permite interação com os controles da interface.
Este código simples abaixo mostra que o evento *DoWork* será executado em background:

```
public static void Main()
{
 System.Threading.ThreadPool.QueueUserWorkItem(new WaitCallback(DoWork));
}
private static void DoWork(object parameter)
{ … }
```

Lembrando que o acesso aos controles de UI por parte do *ThreadPool* é ccomplexo e só poderia ser feito com *call-backs* e *delegates*, o que complica bastante o desenvolvimento. No caso de interação com interface visual, o *Dispatcher* é o indicado. Em caso de processamento com arquivos ou manipulação de variáveis, por exemplo, o *TheadPool* é mais simples de ser codificado e não gera bloqueio ao controle como acontecer com o *Dispatcher*. Um detalhe importante é que o *ThreadPool* não pode ultrapassar 250 processos na fila.

Processos assíncronos – BackgroundWorker

Diferente do *Dispatcher* e *ThreadPool* este controle é gráfico no WinForms e pode manipular controles da interface. Seu funcionamento é similar ao *ThreadPool* com o evento *DoWork.*

Processo assíncronos – TPL (Task Parallel Linq)

Esta classe permite executar processos assíncronos quando se deseja criar uma série de processos, como o exemplo abaixo:

```
foreach (var item in source)                //processo normal, sequencial
        DoSomething(item);
Parallel.ForEach(source, item => DoSomething(item));          //processo paralelo (threads)
```

Note que o primeiro fragmento executa um método a cada item, já a segunda instrução cria uma thread para cada processo, baseada em uma pesquisa LINQ.

Localização

Alguns recursos são importantes quando trabalhamos com aplicativos para multiplos idiomas.
Por exemplo, a propriedade dos controles *RightToLeft* permite inverter a escrita para se ajustar a países onde a leitura é da direita para esquerda. Outro recurso é a tag abaixo que indica que o XML está no formato 16 bits (UTF-16) ao invés dos 8 bits (UTF-8) usado normalmente:

```
<?xml version="1.0" encoding="UTF-16" ?>
```

Para quem não conhece, este indica que para cada letra serão usados 16 bits (2 bytes) ao invés dos 8 bits (1 byte) para nosso alfabeto que só suporta 256 variações, enquanto outros idiomas tem muito mais que isso, dobrando a possibilidade para mais de 32 mil variações de caracteres.
Outro exemplo é a tag *xml:lang="en-US"* que ao ser colocada em um objeto ou controle indica em que idioma está atualmente aquele objeto e seus filhos, e afetará como deverá ser mostrado números, separadores e datas.
O código abaixo é um outro exemplo, onde podemos saber a opção de pais que o usuário escolheu (Painel de Controle do Windows), o idioma do aplicativo e do Windows e criar um objeto para utilizar formatadores e outros recursos específicos, como por exemplo a função *NumberFormat*:

```
CultureInfo X = new CultureInfo("pt-BR");          //Cria uma nova cultura
MessageBox.Show(X.DisplayName);
MessageBox.Show(CultureInfo.CurrentCulture.DisplayName);      //Mostra o idioma do CP
MessageBox.Show(CultureInfo.CurrentUICulture.DisplayName);  //Mostra o idioma da interface
```

Mais detalhes sobre estes recursos podem ser acessados em http://msdn.microsoft.com/en-us/library/ms788718.aspx

Globalização (processo manual)

O processo de globalização é o mesmo utilizado desde outras versões, onde se cria um arquivo "resx" com o nome do idioma desejado, por exemplo "Resources.pt-BR.resx" e "Resources.en-US.resx" onde o primeiro indica o idioma e o segundo parâmetro o pais.
Para isso copie o arquivo "Resource.resx" e renomei acrescentando o pais e idioma. Ao dar duplo clique nestes arquivos é possível criar uma lista de valores para as strings desejadas.
Para utilizar estas strings em idiomais segue o código de exemplo:

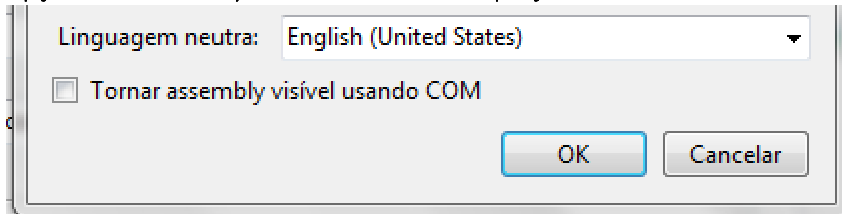
```
<Window ...
     xmlns:props="clr-namespace:WpfApplication1.Properties"
     ...>
 ...
 <Label Content="{x:Static props:Resources.NomeBotaol}" />
```

Veja abaixo como utilizer o aplicativo LocBAML para automatizar esta tarefa.

Aplicativo LocBAML para localização

O arquivo BAML é um XML em formato binário que cria automaticamente as tabelas com os textos para cada idioma. Para isso siga a sequencia:

1. Incluia no seu aplicativo o idioma em que foi desenhado originalmente com as tags *<UICulture>en-US</UICulture>* no arquivo "*.csproj*". Isto também pode ser feito pedir propriedades do projeto e nas opções de assembly colocar o idioma do projeto:



2. Compile a aplicação e note que no diretório *bin* será criado um subdiretório com cada idioma, o que você indicou no assembly e o seu atual
3. Baixe o aplicativo em http://go.microsoft.com/fwlink/?LinkID=160016
4. Execute o aplicativo e gere a tabela CSV:
   LocBaml.exe /parse en-US/WpfApplication1.resources.dll /out:Textos.csv
5. Abra o CSV no Excel e customize a tabela. Na sequencia salve o novo arquivo e utilize o comando abaixo para criar o assembly no idioma:
   LocBaml.exe /generate en-US/WpfApplication1.resources.dll /trans:Texto-pr.csv /out:pt-BR /cul:pt-BR
6. Note que já foi criado o diretório e o assembly para o novo idioma, basta copiar a dll nova para lá
7. Dentro da sua aplicação execute o código abaixo para testar se o idioma foi alterado:

```
CultureInfo culture = new CultureInfo("pt-BR");
Thread.CurrentThread.CurrentCulture = culture;
Thread.CurrentThread.CurrentUICulture = culture;
```

Para mais informações e o passo-a-passo detalhado veja em http://msdn.microsoft.com/en-us/library/ms746621(VS.85).aspx#build_locbaml

Drag-and-Drop

As operações de arrastar e soltar são bem conhecidas e o primeiro código é iniciar a operação de Drag:

```
private void Texto_PreviewMouseRightButtonDown(object sender, MouseButtonEventArgs e)
{
   DragDrop.DoDragDrop(Texto, Texto.Content, DragDropEffects.Copy);
}
private void Texto_PreviewMouseMove(object sender, MouseEventArgs e)
{
   DragDrop.DoDragDrop(Texto, Texto.Content, DragDropEffects.Copy);
}
```

O passo seguinte é utilizar o evento Drop do objeto que irá receber o dado, que deve ter a propriedade *AllowDrop* habilitada, como no caso abaixo onde foi arrastado o conteúdo do *label* acima para outro:

```
label1.Content = e.Data.ToString();
```

Veja mais detalhes desta operação em http://msdn.microsoft.com/en-us/library/ms742859.aspx

Segurança - Security Restriction Policy (SRP)

Este modelo de segurança era anteriormente configurado pelo Painel de Controle e hoje foi delegado a ambientes utilizando *Software Restriction Policies* que podem ser feitas no computador local (gpedit.msc) ou por regras de domínio da rede (GPO's).

Segurança – Code Access Security (CAS)

Este modelo também implementado anteriormente via Painel de Controle agora se tornou transparente e não precisa mais ser manipulado. Detalhes sobre as mudanças no CAS podem ser vistas em http://msdn.microsoft.com/en-us/library/ff527276.aspx e um exemplo onde se utiliza o CAS para testar permissões segue abaixo:

```
public void Save()
    {
      if (IsPermissionGranted(new FileIOPermission(FileIOPermissionAccess.Write, @"c:\newfile.txt")))
```

```
        {
          FileStream stream = File.Create(@"c:\newfile.txt");
          StreamWriter writer = new StreamWriter(stream);
        }
        else
          MessageBox.Show("Permissões insuficientes.");
    }

    bool IsPermissionGranted(CodeAccessPermission requestedPermission)
    {
        try
        {
          requestedPermission.Demand();
          return true;
        }
        catch
        {        return false;        }
    }
```

Note que o código a seguir primeiro pede a permissão por demanda para gravar arquivos (*FileIOPersmissionAccess.Write*) e testa se está ou não garantida. Porem, não é como antes que nos atributos se fazia o pedido por demanda e o motivo é que o UAC do Windows Vista em diante não permite permissões por demanda, estas tem que ser feitas no momento da execução por utilizar o "Executar como Administrador" do Windows. Porem, o método acima serve para testar se o UAC liberou ou não a operação no local indicado.

Segurança – UAC

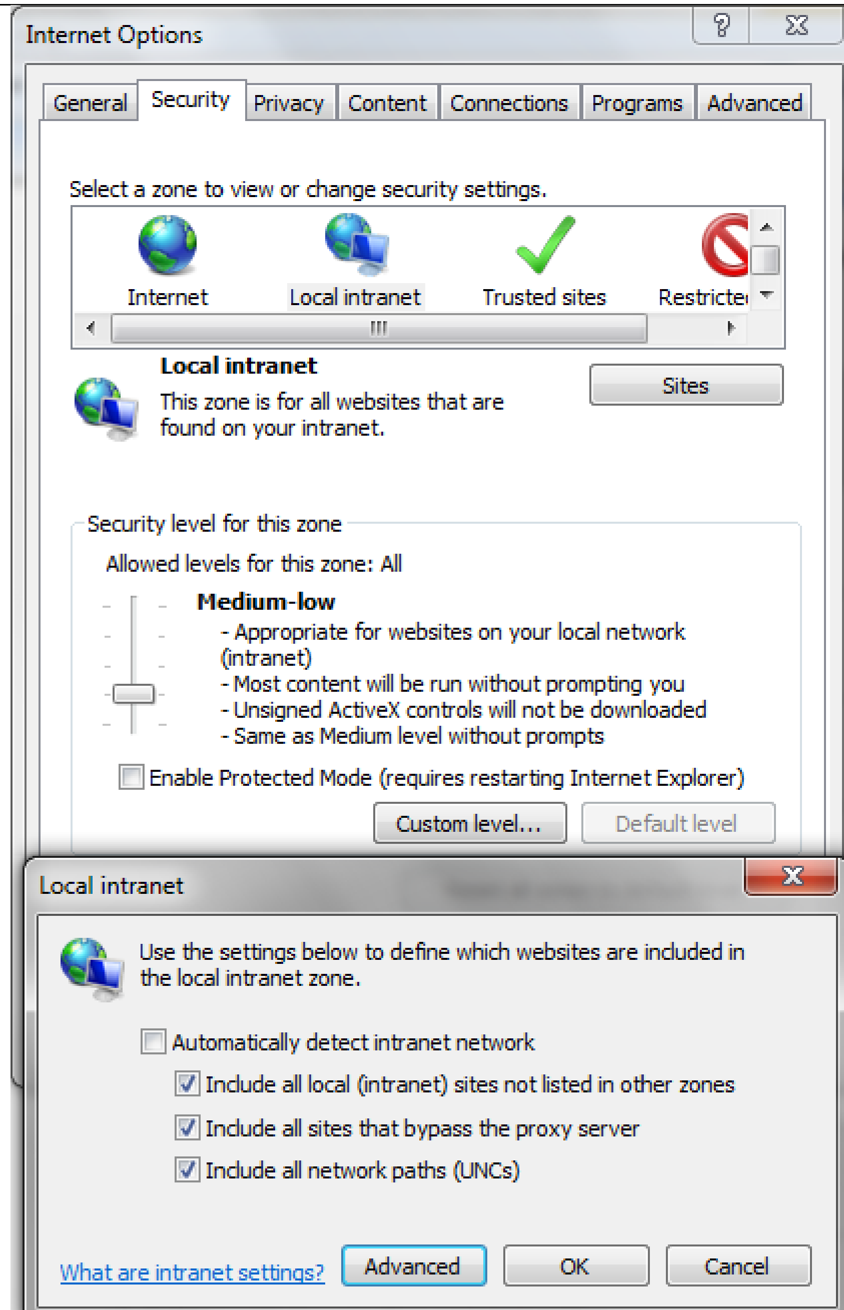
O Windows Vista introduziu um novo conceito de segurança onde o programa é executado no contexto do usuário por padrão e para ter acesso ao raiz ou diretórios especiais do discos, registry e outras opções é necessário utilizar o "Executar como Administrador" do Windows.
Este recurso é importantíssimo e alterou em muito a forma de programar e acessar objetos, como já colocado no tópico acima. Mais detalhes, o artigo "Como desenhar sua interface para o UAC" em http://msdn.microsoft.com/en-us/library/bb756990.aspx descreve bem este novo cenário.

Segurança – Partial and Full Trust

Como parte do novo conceito de CAS transparente, agora as aplicações utilizam o chamado *Partial Trust ou Full Trust*, que nada mais é do que as opções de local utilizadas no próprio navegador.
Isto tem muito a ver com o fato das aplicações WPF serem executadas em modo seguro e dentro de uma *sandbox*. E se for necessário alterar as permissões parciais? Acesse o link http://msdn.microsoft.com/en-us/library/aa970906.aspx e verá que existem chaves de registry a serem alteradas para isso.
Aplicações executadas como WPF e que rodem localmente na máquina do usuário receber o *Full Trust*, claro que limitado as restrições do UAC, pois executam no contexto de segurança da zona *Local Computer*.
Já aplicações baseadas em browser (*page*) são executadas em *Partial Trust* e contam com uma série de restrições, indicadas pelo navegador pelas permissões da zona *Internet*.
Você pode usar diretivas para indicar o tipo de permissão desejada, por exemplo, a mais comum está abaixo e indica que por padrão o assembly não irá expor como APTCA, ou seja, só se for uso explicito pela aplicação *host* que está utilizando o assemblie utilizar APTCA:

```
[assembly:AllowPartiallyTrustedCallers(PartialTrustVisibilityLevel=NotVisibleByDefault)]
```

Mais detalhes em http://msdn.microsoft.com/en-us/magazine/ee677170.aspx
Porem, uma aplicação pode receber mais permissões por ser colocada na *Intranet Zone* por se configurar os locais seguros pelo navegador, como mostra as opções abaixo tiradas do IE8:



Uma lista completa das operações que podem ou não ser feitas no modo de segurança parcial estão descritas em http://msdn.microsoft.com/en-us/library/aa970910.aspx e precisam ser lembradas no momento do exame, pois são importantes em algumas questões.

Application and User Settings

Muitas vezes precisamos guarder dados fixos para utilizar em uma aplicação.

Uma das formas de fazer isso é criando no *aplicação.exe.config* com a sintaxe a seguir:

```
<configuration> ....
<userSettings>
  <WpfApplication1.Properties.Settings>
    <setting name="Usuario" serializeAs="String">
      <value>Anonimo</value>
    </setting>
  </WpfApplication1.Properties.Settings>
 </userSettings>
 <applicationSettings>
  <WpfApplication1.Properties.Settings>
    <setting name="Maquina" serializeAs="String">
      <value>Indefindo</value>
```

```
            </setting>
          </WpfApplication1.Properties.Settings>
        </applicationSettings>
      </configuration>
```

Essas variáveis ficam disponíveis no sistema e podem ser consultadas a qualquer momento, como o exemplo abaixo, mas note que como as variáveis são fixas só podem ser acessadas para leitura:

```
textBlock2.Text = Properties.Settings.Default.Usuario + " em " + Properties.Settings.Default.Maquina;
```

Pode-se criar também propriedades customizadas utilizando-se da herança com a classe *Settings*:

```
public class Propriedades : ApplicationSettingsBase
{
   [UserScopedSetting()]
   [DefaultSettingValue("Anonimo")]]
   public string Usuario
   {
      get { return this["Usuario"].ToString(); }
      set { this["Usuario"] = value; }
   }

   [ApplicationScopedSetting()]
   public string Maquina
   {
      get { return this["Maquina"].ToString(); }
      set { this["Maquina"] = value; }
   }
}
```

Note que neste caso é necessário criar uma classe para fazer a persistência que pode ser criada utilizando métodos que o próprio usuário deseje ou herdando as classes *LocalFileSettingsProvider*. Veja também que definimos um valor padrão para a propriedade.

O acesso a estas propriedades se dá como no caso de uma classe comum, onde instancia-se a classe e utiliza-se suas propriedades:

```
Propriedades Prop = new Propriedades();
Prop.Usuario="Marcelo"; Prop.Maquina = "Notebook";
textBlock2.Text = Prop.Usuario + " em " + Prop.Maquina;
```

Mas vale ressaltar novamente que você precisa criar algum método de persistência.

Tópico 5 – Estabilização e Deploy

Testes

Existem vários tipos de testes que podem ser implementados pelo Visual Studio 2010, os mais comuns são:

- Unit test - Testes com métodos da sua aplicação
- Coded UI test - Cria testes para os controles da interface visual, permitindo gravar a sequencia de ações do usuário e reproduzindo o comportamento
- Database unit test – Teste de stored procedures, functions ou triggers do banco de dados.

Unit Test

Este é o tipo de teste mais comum e faz a execução de uma classe e com os comandos *Assert* geram uma lsita de saída. A classe de teste pode ser criada manualmente ou apenas clicando em um método com o botão direito utilizar "Criar testes unitários" e selecionar os métodos desejados.

Segue abaixo um exemplo de classe de testes, neste caso criada manualmente:

```
[TestClass]
public class UnitTest1
{
   private TestContext testContextInstance;

   [TestMethod]
   public void TestaMetodo()
   {
      WpfApplication1.Propriedades Prop = new WpfApplication1.Propriedades();
      Prop.Usuario = "Marcelo";
      Assert.AreEqual("Marcelo", Prop.Usuario);
   }
```

Ao executar o teste é possível saber se ocorreu com sucesso, baseado na resposta da ultima linha no método acima, como mostra o resumo da figura abaixo:

Teste de interface gráfica

Este teste utiliza o Microsoft UI Automation, série de recursos que permite interagir por código com os comportamentos de um controle, como por exemplo, clicar sobre um listbox, alterar o estado de um checkbox e diversos outras possibilidades.

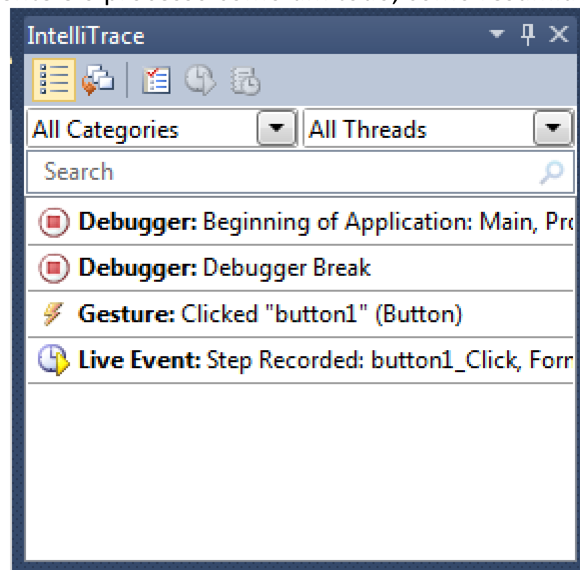
O exemplo abaixo mostra isso procurando o controle MeuBotão e simulando uma invocação, no caso de botão representado pelo clique:

```
AutomationElement Botao = AutomationElement.RootElement.FindFirst(
 TreeScope.Descendants,  new PropertyCondition(  AutomationElement.AutomationIdProperty,
      "MeuBotao"));
InvokePattern pattern = (InvokePattern) Botao.GetCurrentPattern(InvokePattern.Pattern);
pattern.Invoke();
```

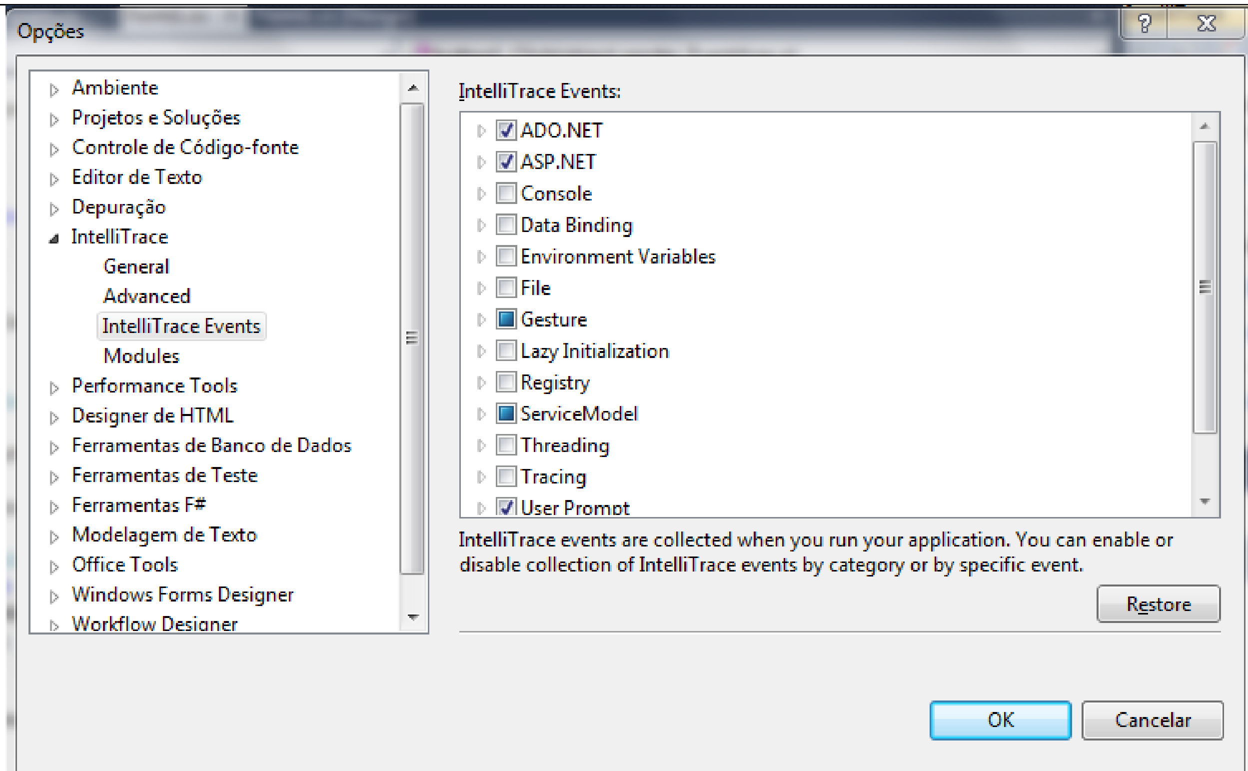
Basicamente os *patterns* possiveis são *Invoke* como o clique do botão e o clique em uma lista, *Toggle* que altera a situação como um checkbox ou menu de opções e *ExpandCollapse* para expandir menus e árvores.

Debug - IntelliTrace

Este recurso permite ver e depurar a execução de um programa de forma mais rica do que um debug normal linha e linha com o uso de *watchs* já que ele permite selecionar a *thread*, saber o "gesto" que o usuário executou, em que evento estamos atualmente e o processo como um todo, como resumido abaixo:

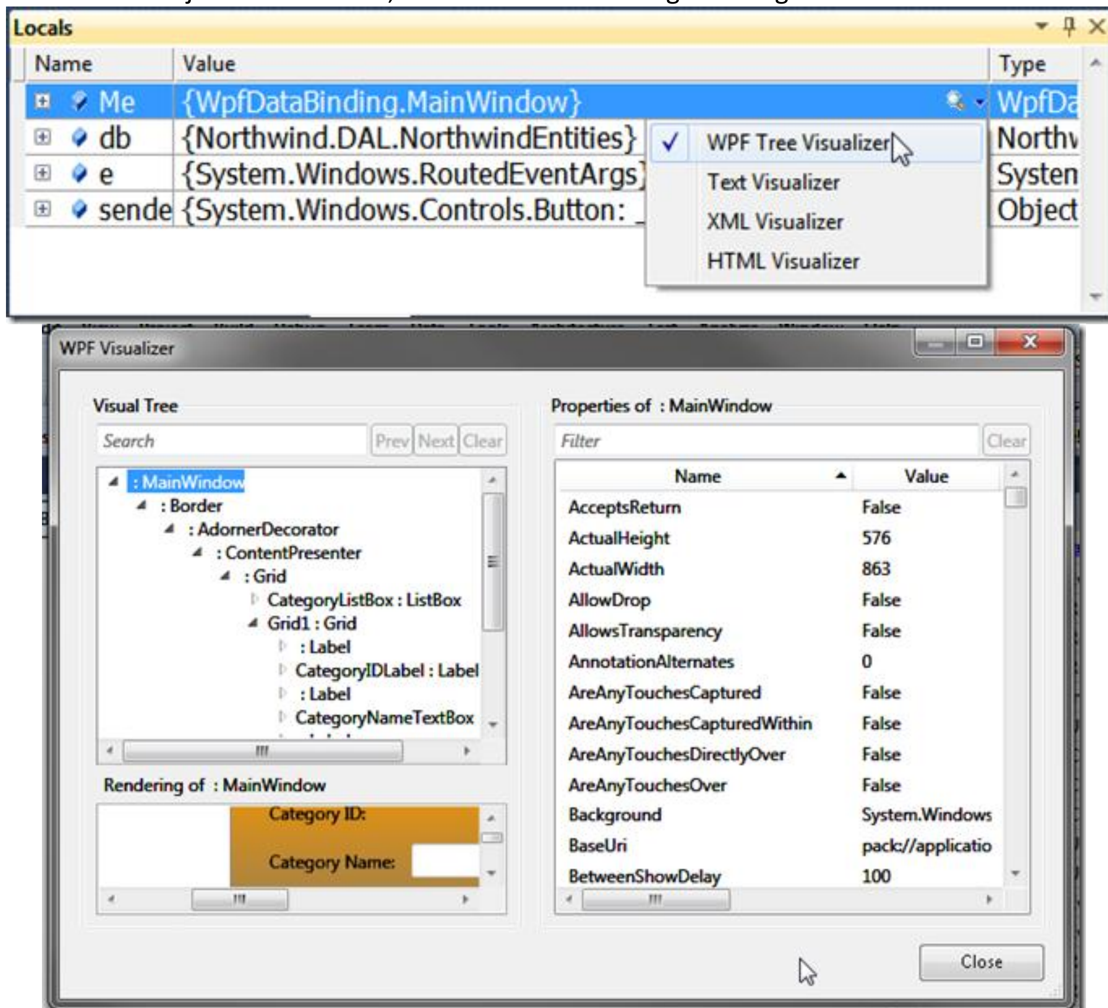


O IntelliTrace permitirá fazer o debug completo e utilizando as opções pode ser configurado para mostrar em detalhes o que o programador precisa ver:



Debug - WPF Visualizer

Ele pode ser chamado a partir das janelas de depuração *Local, Watch* ou *Autos* e permite ver as propriedades, valores e detalhes dos objetos de um WPF, como mostram as imagens a seguir:



Para mais detalhes sobre os métodos de debug do WPF veja http://msdn.microsoft.com/en-us/library/dd409908(VS.100).aspx

Debug - PresentationTraceSources

Esta classe permite que o Visual Studio gere mensagens detalhadas na janela *Output* para propriedade como *Binding* e seus derivado, bem como das classes *DataSourceProvider*. O exemplo abaixo mostra um *binding* com alto nível de tracing, pode ser *None, Low, Medium* e *High*:

```
<Window
 ...
 xmlns:diag="clr-namespace:System.Diagnostics;assembly=WindowsBase">
 ...
 <TextBox>
  <TextBox.Text>
   <Binding Path="Name"
        diag:PresentationTraceSources.TraceLevel="High"/>
  </TextBox.Text>
 </TextBox>
</Window>
```

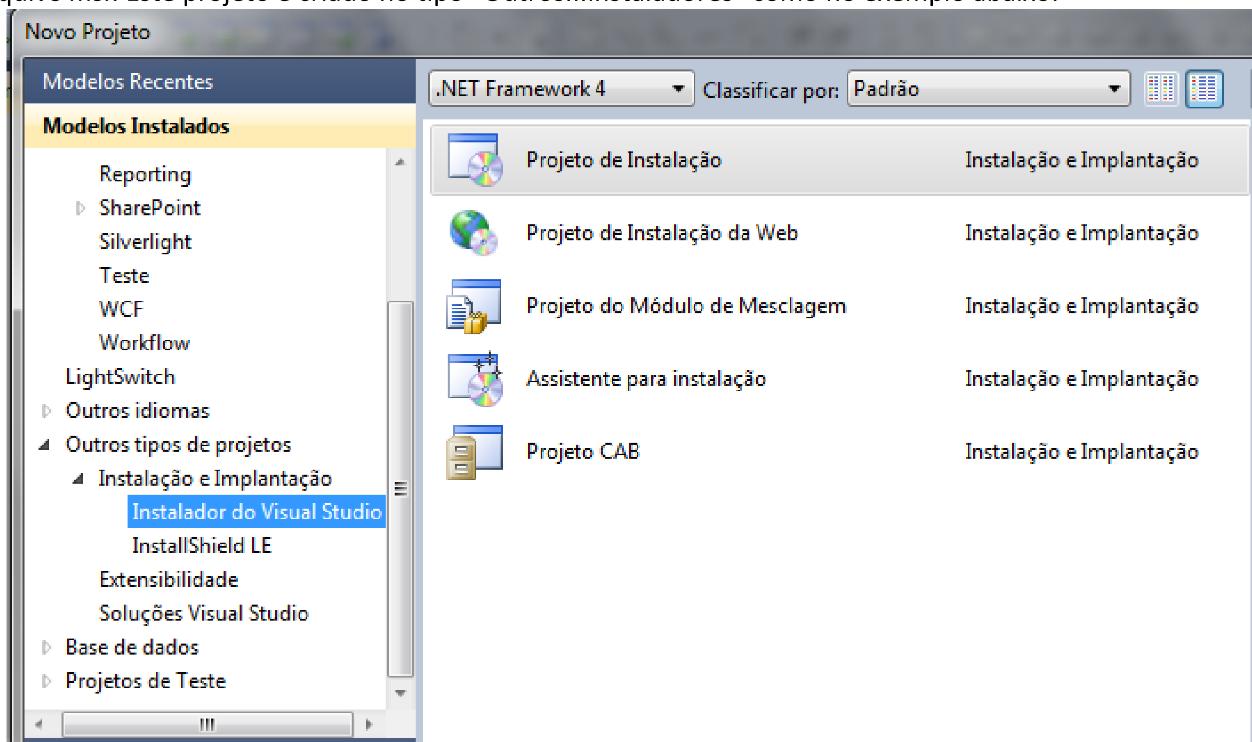
Mais detalhes em http://msdn.microsoft.com/en-us/library/system.diagnostics.presentationtracesources_attachedproperties(VS.100).aspx

Deploy – XCOPY

Esta é a forma mais simples de deploy em um ambiente interno. Basta copiar os arquivos gerados no diretório *bin\release* da aplicação e o programa já pode ser executado no cliente.
Apesar de simples este método não gera atalhos, não organiza os arquivos e não é automatizado. Ainda outra limitação é que o usuário precisará já ter instalado o .NET Framework 4.

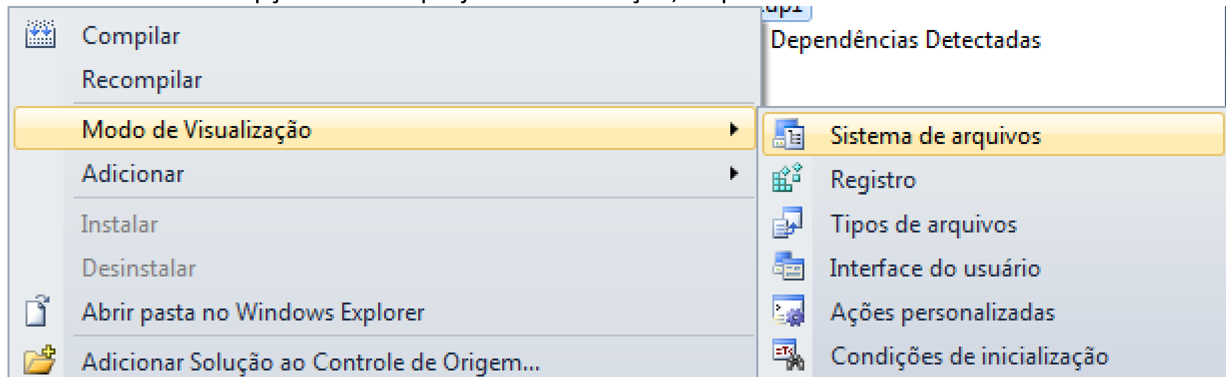
Deploy - Windows Installer Project

Uma das formas mais comuns de deploy de aplicações WinForms, WPF ou XBAP é utilizando um projeto que gere o arquivo *msi*. Este projeto é criado no tipo "Outros...Instaladores" como no exemplo abaixo:



O arquivo msi pode ser executado diretamente em máquinas com .NET Framework 4 e funcionam simular a um *setup.exe* anteriormente muito utilizado.
A vantagem deste método é que automaticamente será criado a estrutura de diretórios, atalhos, desinstalador e outras configurações que você poderá acrescentar como adicionais nas definições do projeto.
Outra vantagem é que o instalador automaticamente irá fazer o download e instalação do .NET Framework 4 a partir da internet se o usuário não tem esta versão instalada.

Além disso, o instalador permite criar ações e condições, por exemplo, só instalar a aplicação se o computador destino contem determinado arquivo, chave de registry ou aplicativo. Ou então criar ações customizadas, como por exemplo, fazer uma confirmação, executar programas externos e outros.
Para acessar essas novas opções em um projeto de instalação, clique sobre ele com o botão direito:



Deploy – ClickOnce

Este é um método de deploy muito bom por permitir a publicação da aplicação como um fonte.
Ele pode ser configurado para todas as vezes em que a aplicação for chamada comparar o manifesto que atualmente está no computador com o que está no diretório de origem e instalar a nova versão, desde que no momento da publicação deve ser selecionada a opção correspondente.
Com este recurso o programador faz o *Build...Publish* do menu do Visual Studio e gerará este instalador. Nas próximas versões o programador chamará novamente o *publish* e irá sobrepor os arquivos que estão no diretório indicado. Automaticamente os usuários que chamarem a aplicação irão instalar a nova versão.
É um processo simples, rápido e essencial para evitar redistribuição da aplicação quando de alguma manutenção.
Para fazer alguma configuração do *ClickOnce* vá nas propriedades do projeto e altere as opções da aba *Publish*.

Arquivo manifesto

O arquivo de manifesto descreve a aplicação, sua necessidade de segurança, a versão e outros dados.
Este arquivo é gerado no momento da compilação e pode ser alterado manualmente por ser um XML.
Porem, o Visual Studio agora inclui uma ferramenta para customizar este arquivo: *Mage.exe* que faz as alterações por parâmetros em linha de comandos e o *MageUI.exe* que permite a manipulação gráfica, como a seguir: