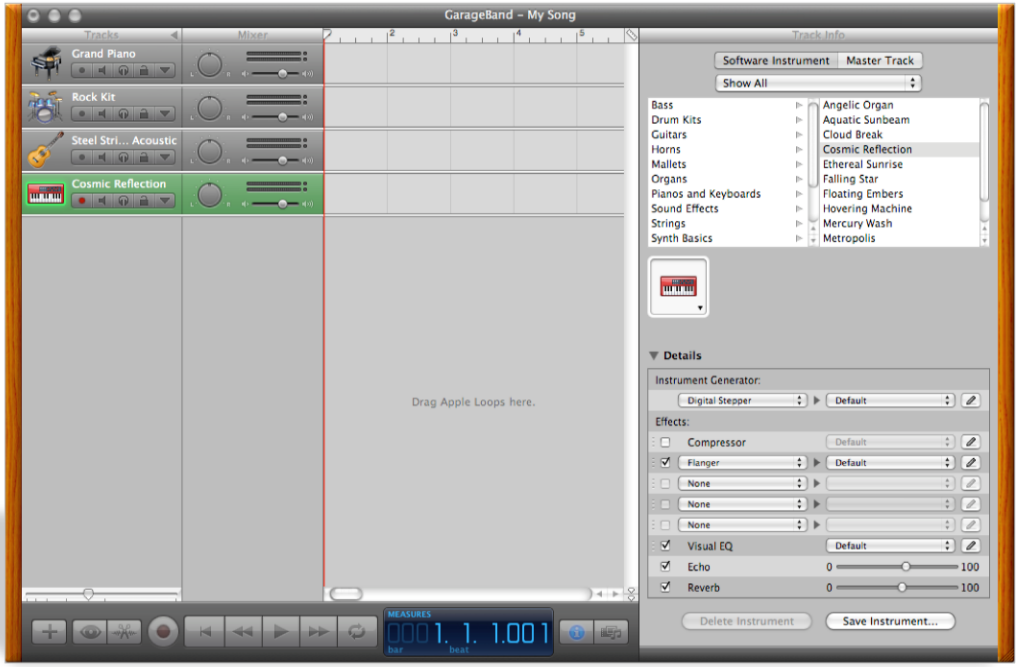
Custom Controls

DEVELOPMENTOR

DEVELOPING PEOPLE WHO DEVELOP SOFTWARE

[Motivation] "Batteries not included"

- Sometimes …
  - control you need just isn't there
  - you need to reuse a section of your UI in other applications
  - changing the visual appearance just isn't enough

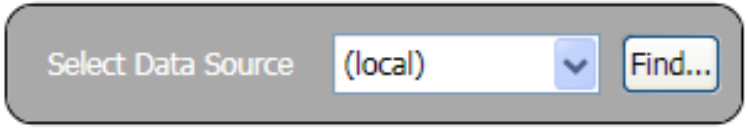


what would it take to build this UI?

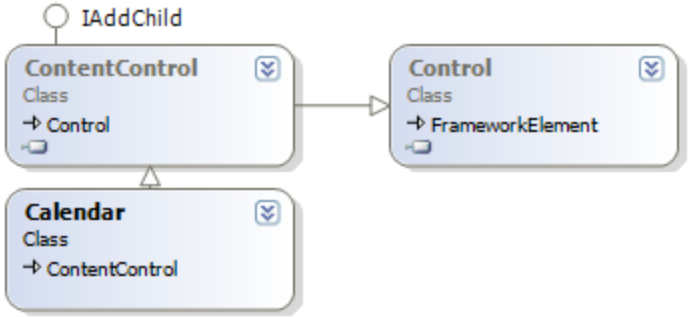
Four ways to customize controls

change basic appearance with a `Style`

replace visual tree completely with a `ControlTemplate`



compose new control with `UserControl`



add / change functionality using inheritance

When should you consider custom controls?

- Do *not* use custom controls for data visualization
  - that is what data templates are for!
- Do *not* use custom controls to alter control appearance
  - that is what control templates are for!
- Do *not* use custom controls to enforce layout
  - that is what panels are for!
- Remember that you can also use composition and inheritance to create new controls based on existing ones
  - **`CheckBoxList`** is unnecessary – just add checkboxes
  - **`FontDialog`** could be easily created through composition

Deciding between UserControl and full control

- Custom controls allow full visualization customization
  - replacement of the **`ControlTemplate`** for the control
- Consider the `UpDownControl`
  - easily creatable through composition as **`UserControl`**

`TextBlock + ScrollBar`

what if the developer wants one instance to look like this instead?

Creating controls from scratch

- Creating a control means implementing both visual appearance and behavior
  - significantly more work and investment
  - gives full control over everything
  - required if you want to theme or allow templating of the control
- Remember you can override existing controls too
  - useful if the control is "almost right"
  - almost all exposed events have virtual **`OnXXX`** method

Steps to create custom controls

1. Choose the base class
   - Visual Studio "WPF Custom Control" template uses **`Control`**
2. Provide the rendering + layout support
   - **`OnRender + MeasureOverride + ArrangeOverride`**
   - .. or supply default **`Control.ControlTemplate`**
3. Provide support for hosting visual children if necessary
   - **`VisualChildrenCount`**
   - **`GetVisualChild`**
   - for single element just use base class **`ContentControl`**

Choosing the base class

- `Control` is useful to create interactive elements
  - has focus support
  - typically composes multiple elements together
  - template support for custom themes
  - **`ContentControl`** supports a user-controlled child
- `FrameworkElement` is used to create raw shapes
  - requires taking over rendering
  - less overhead than full control
  - generally faster than full control
  - **`FrameworkContentElement`** supports user-controlled child
- Also consider built-in base classes for specific types
  - **`Shape`**, **`RangeBase`**, **`ButtonBase`**, **`Selector`** etc.

Things to keep in mind when building controls

- Controls are fully instantiated in design tools
  - check using **`DesignerProperties.IsInDesignMode`**
  - control should not *do anything* when in designer mode
- Use `DependencyProperty` to expose data
  - make sure to set appropriate metadata during creation
  - attach to existing properties to inherit existing system behavior
- Use `RoutedEvents` and `Commands` for events
  - attach events in code behind to child elements
  - reuse existing events from base or even other elements
- Control should participate in system features
  - support templates for applications to redefine appearance
  - changing visual appearance based on current OS theme

Supplying user controllable behavior

- Use public properties and methods to expose behavior
  - should always be backed by **`DependencyProperty`**

```
public partial class Calendar
{
   public readonly static DependencyProperty
                          IsHeaderVisibleProperty = ...
   public bool IsHeaderVisible
   {
      get { return (bool)
            GetValue(IsHeaderVisibleProperty); }
      set { SetValue(IsHeaderVisibleProperty, value); }
   }
}
```

Override inherited undesirable behavior

- Disable or change existing `DependencyProperty` behavior
  - always done in static constructor

```
public partial class Calendar
{
   static Calendar()
   {
      ToolTipService.IsEnabledProperty.
          OverrideMetadata(typeof(Calendar),
            new FrameworkPropertyMetadata(null,
               new CoerceValueCallback(
                      MyCoerceToolTipIsEnabled)));
   }
}
```

turn off tooltips when hovering over certain areas of control by overriding the `ToolTipService.IsEnabled` property for our control

Provide handlers for input events

- Add handler at the class level to ensure it is always received
  - can also process "**Handled**" events

```
public partial class Calendar
{
   static Calendar()
   {
      EventManager.RegisterClassHandler(
         typeof(Calendar),
         Mouse.MouseDownEvent,
         new MouseButtonEventHandler(OnMouseButtonDown),
         true);
   }
}
```

Supporting templates in custom controls

- Should support templates to allow overriding visual style
  - give names to internal elements using **`PART_xxx`** convention
- Supply all visual behavior as part of template
  - mouse-over effects
  - focus effects
  - "press" effects
  - …
- Can define multiple templates for different parts of control
  - header vs. footer vs. body
  - consider providing properties to replace templates
- Modify existing templates from built-in controls to save time
  - Reflector and Blend can both pull templates out

Hooking procedural behavior up to templates

- Override `OnApplyTemplate` to get access to visual children
  - use **`FindName`** to retrieve element defined in template
  - wire up event handlers in code behind

```
public partial class Calendar : Control
{
   public override void OnApplyTemplate()
   {
      base.OnApplyTemplate();

      Button child = (Button) this.Template.FindName(
                  "PART_PrevButton", this);
      if (child != null)
         child.Click += prevMonthButton_Click;
   }
}
```

should not assume template part is supplied

Integrating with the designer

- Supply `[TemplatePart]` to give hints to designers and tools
  - helps template authors know what you need
  - will be used by Blend in the future to identify required parts

uses most basic type possible for extensibility

```
[TemplatePart(Name="PART_PrevButton", Type=typeof(ButtonBase))]
[TemplatePart(Name="PART_NextButton", Type=typeof(ButtonBase))]
[TemplatePart(Name="PART_Header", Type=typeof(FrameworkElement))]
[TemplatePart(Name="MonthTemplate", Type=typeof(FrameworkElement)]
[TemplatePart(Name="YearTemplate", Type=typeof(FrameworkElement)]
public partial class Calendar
{
   ...
}
```

defines visual template for each possible view

Defining states for the Visual State Manager

- `[TemplateVisualState]` used to define states and groups
  - all known visual states should be defined
  - used by Blend to drive VSM support
  - not currently defined on any existing WPF controls (but will be)

```
[TemplateVisualState(Name="Disabled", GroupName="CommonStates")]
[TemplateVisualState(Name="Unfocused", GroupName="FocusStates")]
[TemplateVisualState(Name="MouseOver", GroupName="CommonStates")]
[TemplateVisualState(Name="Focused", GroupName="FocusStates")]
[TemplateVisualState(Name="Normal", GroupName="CommonStates")]
public partial class Calendar
{
  ...
}
```

More designer integration goodness

- Attributes also used to direct property explorer
  - applied to class definition and property accessors

event to create when double-clicked

property visible?

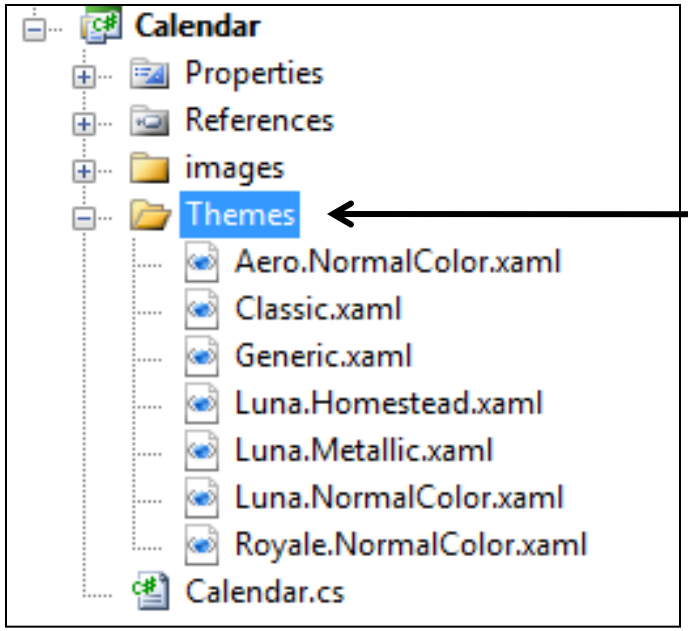
bindable?

where and what to display in property explorer

```
[DefaultProperty("Text")]
[DefaultEvent("OnChange")]
partial class OutlineText : Shape
{
    [Browsable(true)]
    [Bindable(true)]
    [Category("Appearance")]
    [Description("Sets the font size")]
    [TypeConverter(typeof(FontSizeConverter))]
    [Localizability(LocalizationCategory.None)]
    public double FontSize
    {
        ...
    }
}
```

Integrating with the WPF theme support

- Standard controls change appearance based on OS theme
  - have different control templates loaded at runtime
  - WPF will merge theme resources automatically



themes are just `ResourceDictionary` files contained in the `themes` directory

each file has "theme" specific resources, system defaults to `generic.xaml` if resource or theme is not available

Step 1: create initial theme

- At a minimum, custom controls should define generic theme
  - create **`generic.xaml`** resource dictionary in themes folder holding the default control template(s)

```
<ResourceDictionary xmlns="..." xmlns:me="...">
  <Style TargeType="local:Calendar">
    <Setter Property="Control.Template">
      <Setter.Value>
        <ControlTemplate TargetType="local:Calendar">
           <Grid> ... </Grid>
        </ControlTemplate>
      </Setter.Value>
    </Setter>
  </Style>
</ResourceDictionary>
```

`themes/generic.xaml`

Defining colors and fonts

- Consider using system resources for conformity
  - system colors, fonts
  - standard widths, spacing, timings, etc.
- Accessible through properties on `SystemXXX` classes
  - **`System.Windows.SystemParameters`**
  - **`System.Windows.SystemFonts`**
  - **`System.Windows.SystemColors`**

```
<Style TargetType="local:Calendar">
   <Setter Property="Foreground"
        Value="{x:Static SystemColors.WindowTextBrush}" />
</Style>
```

… or even do a resource lookup with `SystemColors.WindowTextBrushKey`

Step 2: override default style

- Override `DefaultStyleKeyProperty` to indicate style is present
  - done in static constructor
  - associates key to current type instead of **`typeof(Control)`**
  - WPF will then locate proper template in resources

```
public partial class Calendar : Control
{
   static Calendar()
   {
      DefaultStyleKeyProperty.OverrideMetadata(
          typeof(Calendar),
             new FrameworkPropertyMetadata(
             typeof(Calendar)));
   }
   ...
}
```

Step 3: indicate theme support

- Add or Modify `ThemeInfoAttribute` in `AssemblyInfo.cs` to indicate theme support[1]
  - first parameter identifies theme-specific resources
  - second parameter is where generic resources are located
  - can be **`ExternalAssembly`**, **`SourceAssembly`**, or **`None`**

indicates theme specific resources are embedded in assembly

```
[assembly: ThemeInfo(
    ResourceDictionaryLocation.SourceAssembly,
    ResourceDictionaryLocation.SourceAssembly)
]
```

generic resources in `themes\generic.xaml` embedded in custom control assembly

Step 4: adding other themes

- Additional OS themes can also be created – location is identified by `ThemeInfoAttribute`
  - stored either in same assembly (**`SourceAssembly`**)
  - or secondary assembly (**`External`**) to save space
- Dictionary must be named appropriate and placed in themes
  - external theme assemblies must be named specifically as well

Windows 7 or Vista Aero	**`themes\Aero.NormalColor.xaml`**
Windows XP Blue	**`themes\Luna.NormalColor.xaml`**
Windows XP Silver	**`themes\Luna.Metallic.xaml`**
Windows XP Olive Green	**`themes\Luna.Homestead.xaml`**
Windows Media Center	**`themes\Royale.NormalColor.xaml`**
Windows 2003 (Classic)	**`themes\Classic.xaml`**

Utilizing resources located in themes

- String-based key not sufficient to find theme resources
  - really need key <u>and</u> type information to make it unique
- `ComponentResourceKey` used as resource name[1]
  - contains type and object-based key information
  - must be used to both associate key and locate key

```
<ResourceDictionary xmlns="..." xmlns:me="...">

<SolidColorBrush Color="AliceBlue"
     x:Key="{ComponentResourceKey
        {x:Type local:Calendar}, SelectedDateBrush}" />

</ResourceDictionary>
```

themes/generic.xaml

Advertising resources on custom controls

- Expose the key through a property on the custom control
  - can be consumed by the client using **`{x:Static}`** lookup

```
public partial class Calendar : Control
{
   ...
   public static readonly
        ComponentResourceKey SelectedDateBrushKey =
           new ComponentResourceKey(
                 this.GetType(), "SelectedDateBrush");
   ...
}
```

```
<Button Background="{DynamicResource
   {x:Static someAssembly:Calendar.SelectedDateBrushKey}}"
    ... />
```

References

- Control Authoring overview -
  - http://msdn2.microsoft.com/en-us/library/ms745025.aspx
- Creating "lookless" controls
  - http://www.cubido.at/Blog/tabid/176/EntryID/81/Default.aspx
- Guidelines for styling controls
  - http://msdn2.microsoft.com/en-us/library/c52dde45-a311-4531-af4c-853371c4d5f4.aspx
- Color Picker custom control sample
  - http://msdn2.microsoft.com/en-us/library/ms771620.aspx
- Source code for existing WPF controls
  - http://www.codeplex.com/NetMassDownloader

Summary

- `UserControls` allow reuse of composed visual structure
  - provides easy way to split complex XAML into separate files
- Full custom controls can also be built
  - choose the base class depending on requirements
  - more flexible than **`UserControls`**, but also **harder to build**
- Several 3rd party vendors are building controls
  - Infragistics (http://www.infragistics.com)
  - Blendables (http://www.blendables.com)
  - DevComponents (http://www.devcomponents.com)
  - MobiForm (http://www.mobiform.com)
  - Xceed (http://www.xceed.com)
  - DevExpress (http://devexpress.com)