Windows Forms

Custom Controls

Krzysztof Mossakowski
Faculty of Mathematics and Information Science
http://www.mini.pw.edu.pl/~mossakow

Windows Forms Custom Controls

- A composite control (user control)
  - a collection of Windows Forms controls encapsulated in a common container
- An extended control (derived control)
  - derive an inherited control from any existing Windows Forms control
  - extend its functionality by adding custom properties, methods, or other features
  - override the `OnPaint()` method to get a custom appearance
- A custom control
  - inherited from the `Control` class
  - the most powerful way to create a control

Base Classes for Custom Controls

- **`Component`** – can be dragged from the toolbox but it doesn't get a piece of form real estate (e.g. **`ToolTip`**, **`Timer`**, etc.)
- **`Control`** – mouse and keyboard support (owner-drawn controls)
- **`ScrollableControl`** – support for scrolling
- **`ContainerControl`** – support for containing child controls and managing their focus (e.g. **`GroupBox`**, **`Panel`**)
- **`UserControl`** – the **`Load`** event for initialization, design-time support (user controls)
- **`Form`** and other control classes – deriving from the **`Form`** class to create a reusable form template, or deriving from an existing control to override and enhance its functionality (derived controls)

Aspects of Creating Custom Controls

- All standard system behaviour must be mimic manually, e.g.:
  - scrolling support
  - focus cues (i.e. indicating when the control has focus)
  - the "pushed" state appearance for a button control
  - special cues or "hot tracking" appearance changes when the mouse moves over the control
  - hit testing to determine if a click was made in an appropriate area
  - respecting and applying the Windows XP or Vista themes

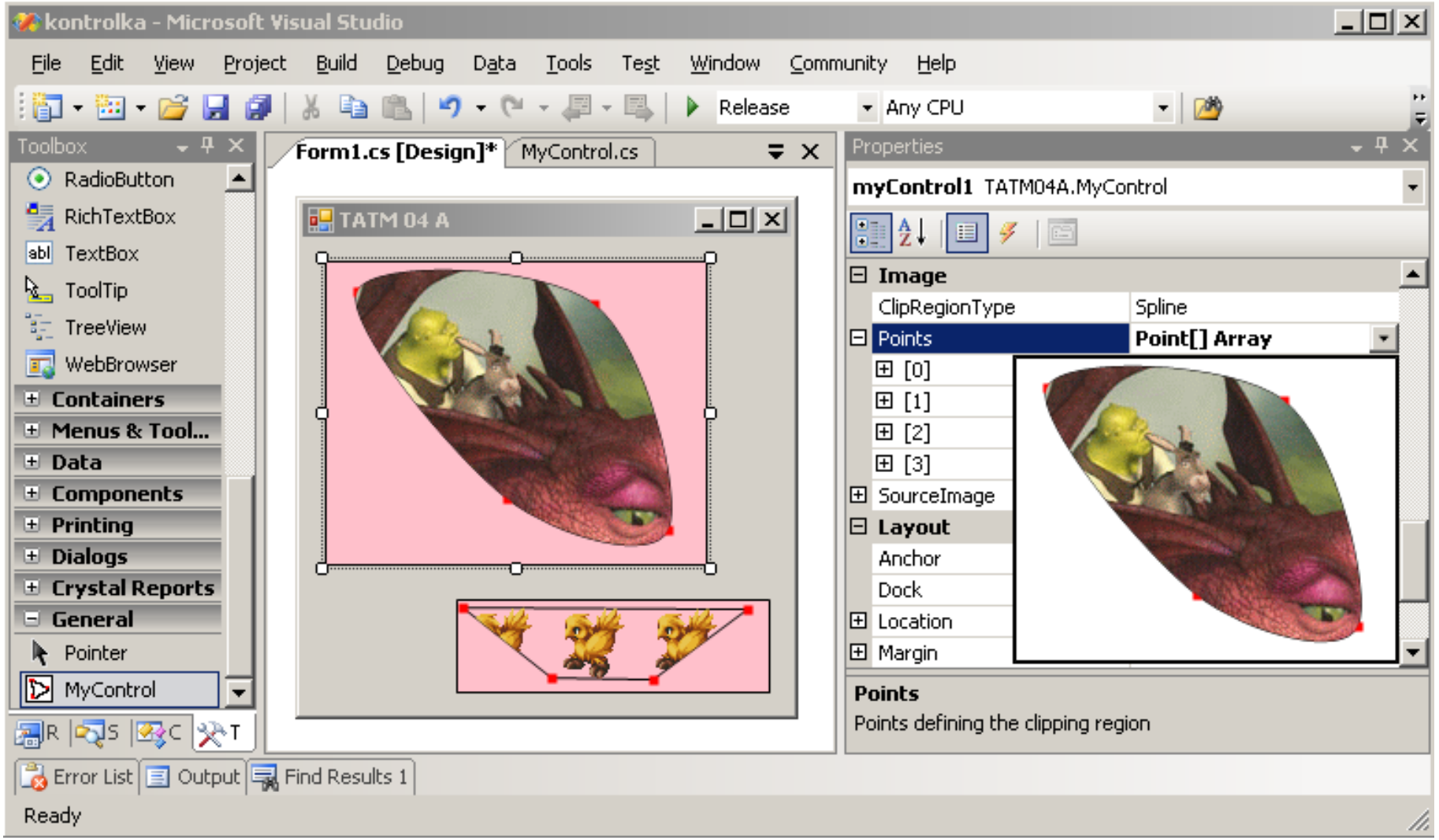
Visual Studio Toolbox Support

- Every time a class library is compiled, Visual Studio scans through the classes it contains, and adds each component or control to a special temporary tab at the top of the Toolbox
- The first time a control is added to a project (e.g. by dragging from the Toolbox), Visual Studio:
  - adds a reference to the assembly where the control is defined
  - copies this assembly to the project directory
- The Toolbox can be customized
  - the Toolbox is a user-specific Visual Studio setting, not a project-specific setting

Design Time Issues

- Allowing the developer to add the control to a form and configure it at design time
- Ensuring the developer's configuration steps are properly serialized into the form code so the control can be successfully initialized when the program is executed
- Ensuring the control behaves nicely at runtime, e.g. a realistic representation of the runtime appearance
- Giving design-time shortcuts for complex configuration tasks (right-click context menus, smart tags, advanced editors for specialized properties, and so on)
- Using licensing to differentiate between development and runtime use of a control, and restricting use according to your license policy

Sample Custom Control



Design Time Support

- Attributes
  - supplying information that will be used in the Properties window
  - attaching other design-time components to the control and configuring how properties are serialized
- Type converters
  - allowing complex or unusual data types to be converted to and from representations in more common data types
  - generating the initialization code required to instantiate a complex type
- Type editors
  - providing a graphical interface for setting complex type values
- Control designers
  - managing the control's design-time appearance and behaviour

Design-Time Attributes

- For classes:
  - **`DefaultPropertyAttribute`** – the property will be selected after clicking on the control
  - **`DefaultEventAttribute`**
- For properties:
  - **`DefaultValueAttribute`**
  - **`EditorAttribute`** – an editor to use by Visual Designer
  - **`LocalizableAttribute`** – the property will be stored in resources when the developer starts localization
  - **`TypeConverterAttributes`**
- For properties and events (appearance in property browser):
  - **`BrowsableAttribute`**
  - **`CategoryAttribute`**
  - **`DescriptionAttribute`**

Implementing a Type Converter

- Derive from the `TypeConverter` class
- Override:
  - `CanConvertFrom()`, `CanConvertTo()` – if the conversion can be done
  - `ConvertFrom()`, `ConvertTo()` – to make a conversion
  - `IsValid()` – to validate
- All these methods are implemented in the `TypeConverter` class, override them if necessary

Example of Type Converter Implementation

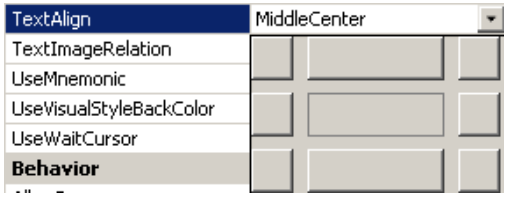
```
public class PointConverter : TypeConverter {

  public override bool CanConvertFrom(
          ITypeDescriptorContext context, Type sourceType) {
    if (sourceType == typeof(string)) {
      return true;
    }
    return base.CanConvertFrom(context, sourceType);
  }

  public override object ConvertFrom(
          ITypeDescriptorContext context,
          CultureInfo culture, object value) {
    if (value is string) {
      string[] v = ((string)value).Split(new char[] {','});
      return new Point(int.Parse(v[0]), int.Parse(v[1]));
    }
    return base.ConvertFrom(context, culture, value);
  }

  public override object ConvertTo(
          ITypeDescriptorContext context, CultureInfo culture,
          object value, Type dstType) {
    if (destinationType == typeof(string)) {
      return ((Point)value).X + "," + ((Point)value).Y;
    }
    return base.ConvertTo(context, culture, value, dstType);
  }
}
```

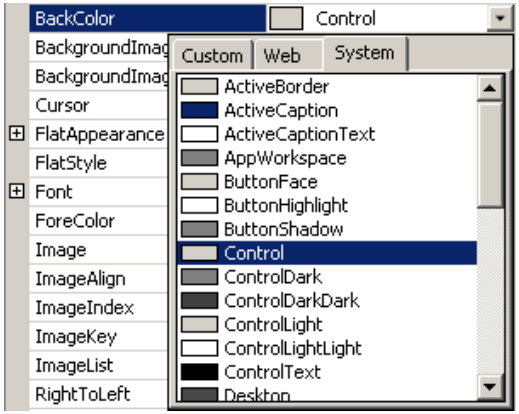
Prebuilt Type Editors

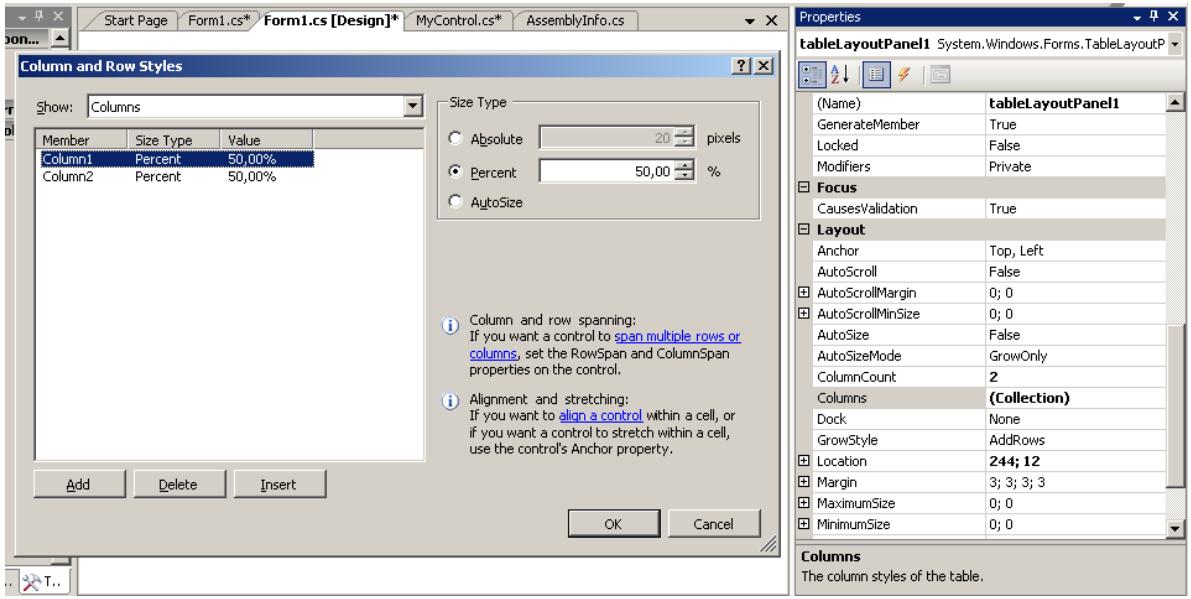
- System.ComponentModel.Design:
  - ArrayEditor, BinaryEditor, CollectionEditor, MultilineStringEditor
- System.Drawing.Design:
  - FontEditor, ImageEditor
- System.Web.UI.Design.WebControls:
  - RegexTypeEditor
- System.Windows.Forms.Design:
  - MaskPropertyEditor, FileNameEditor, FolderNameEditor, ShortcutKeysEditor

Editing Custom Types

- Possibilities:
  - edit as a string - requires a **`TypeConverter`** for a custom type
  - edit with a drop-down UI - requires a **`UITypeEditor`**
  - edit with a modal dialog box - requires a **`UITypeEditor`**







Implementing a UI Type Editor

- Necessaries:
  - define a class derived from **`UITypeEditor`**
  - override **`GetEditStyle()`**, return: **`None`**, **`DropDown`** or **`Modal`**
  - override **`EditValue()`**, parameters:
    - **`ITypeDescriptorContext`** – the context (also the control which is being edited)
    - **`IServiceProvider`** – for displaying a form or a drop-down
- Optional possibilities:
  - a constructor to make initialization
  - **`GetPaintValueSupported()`**, **`PaintValue()`** – displaying value's representation

Example of UI Type Editor Implementation

```
public class MyEditor : UITypeEditor {

  public override object EditValue(
                    ITypeDescriptorContext context,
                    IServiceProvider provider, object value) {
    if (context != null && context.Instance != null &&
        provider != null) {
      IWindowsFormsEditorService edSvc =
            (IWindowsFormsEditorService)provider.GetService(
                          typeof(IWindowsFormsEditorService));
      if (edSvc != null) {
        MyControl orgCtrl = (MyControl)context.Instance;
        MyControl propCtrl = new MyControl();
        propCtrl.Width=orgCtrl.Width;
        (...)
        edSvc.DropDownControl(propCtrl);
        return propCtrl.Points;
      }
    }
    return value;
  }

  public override UITypeEditorEditStyle GetEditStyle(
                          ITypeDescriptorContext context) {
    return UITypeEditorEditStyle.DropDown;
  }
}
```

Code Serialization

- When the control's properties are configured in the Properties window, Visual Studio needs to be able to create the corresponding code statements in the `InitializeComponent()` method of the containing form
- Basic serialization – Visual Studio inspects the public read/write properties of a control and generates the corresponding statements that set them
- The `DefaultValueAttribute` can be used to limit the number of serialized properties – only properties with values diferrent than default values are serialized

Programmatic Code Serialization

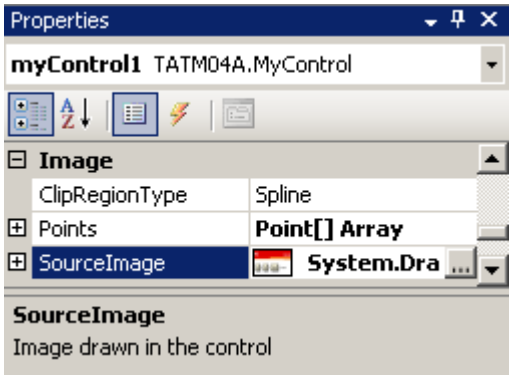
- **`Reset...`** – sets a property to its default value
- **`ShouldSerialize...`** – Visual Studio writes a code to the form only if a property is changed
- Do not use **`DefaultValueAttribute`** if these methods are used

```
[(...)] public Image SourceImage { (...) }

public bool ShouldSerializeSourceImage() {
    return (image != null);
}

[(...)] public Point[] Points { (...) }

public void ResetPoints() {
    points[0].X = 25;  points[0].Y = 25;
    (...)
    Invalidate();
}
```

Creating Custom Properties

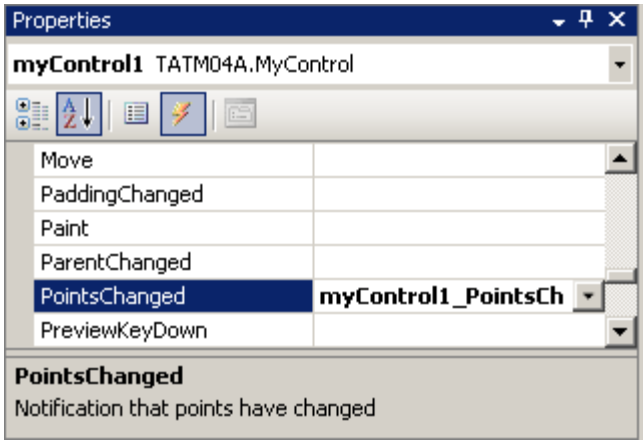
- Apply design-time attributes
- Call `Invalidate()` when the control must be redrawn
- If the property is a custom (nonstandard) data type, type converter must be associated with it

```
[Browsable(true),
Category("Image"),
DefaultValue(null),
Description("Image drawn in the control")]
public Image SourceImage {
  get {
    return image;
  }
  set {
    image = value;
    Invalidate();
  }
}
```



Custom Events

```
private EventHandler onPointsChanged;

[Browsable(true),
Category("Specific events"),
Description("Notification that points have changed")]
public event EventHandler PointsChanged {
    add    {
        onPointsChanged += value;
    }
    remove {
        onPointsChanged -= value;
    }
}

public Point[] Points {
    get { return points; }
    set {
        points = value;
        if (onPointsChanged != null) {
            onPointsChanged(this, EventArgs.Empty);
        }
        Invalidate();
    }
}
```



Control Painting

```
protected override void OnPaint(PaintEventArgs pe) {
  Graphics gr = pe.Graphics;
  gr.SmoothingMode = SmoothingMode.AntiAlias;
  if (DesignMode) {
    gr.FillRectangle(Brushes.Pink, pe.ClipRectangle);
  }
  GraphicsPath path = BuildPath();
  if (DesignMode || EditMode) {
    gr.DrawRectangle(Pens.Black,0,0,Width-1,Height-1);
    gr.DrawPath(Pens.Black, path);
    for (int i = 0; i < points.Length; i++) {
      gr.FillRectangle(Brushes.Red,
                       Points[i].X-size,Points[i].Y-size,
                       2*size+1,2*size+1);
    }
  }
  if (this.image != null) {
    TextureBrush tb = new TextureBrush(image);
    gr.FillPath(tb,path);
    tb.Dispose();
  }
  base.OnPaint(pe);
}
```

Using Visual Styles

- The `ControlPaint` class – rendering common Windows Forms controls
- Classes designed to draw the related control regardless of whether visual styles are available:
  - `ButtonRenderer, CheckBoxRenderer, GroupBoxRenderer, RadioButtonRenderer`
- Classed designed only to use visual styles:
  - `ComboBoxRenderer, ProgressBarRenderer, ScrollBarRenderer, TabRenderer, TextBoxRenderer, TrackBarRenderer`

Using a Control Rendering Class

```
protected override void OnPaint(PaintEventArgs e)
{
  base.OnPaint(e);

  if (!ComboBoxRenderer.IsSupported) {
    ControlPaint.DrawComboButton(e.Graphics,
                                 this.ClientRectangle,
                                 ButtonState.Normal);
  } else {
    ComboBoxRenderer.DrawDropDownButton(e.Graphics,
                                 this.ClientRectangle,
                                 ComboBoxState.Normal);
  }
}
```

Using a Visual Style Element

```
private VisualStyleRenderer renderer = null;
private readonly VisualStyleElement element =
    VisualStyleElement.StartPanel.LogOffButtons.Normal;

public CustomControl()
{
  if (Application.RenderWithVisualStyles &&
      VisualStyleRenderer.IsElementDefined(element)) {
    renderer = new VisualStyleRenderer(element);
  }
}

protected override void OnPaint(PaintEventArgs e)
{
  if (renderer != null) {
    renderer.DrawBackground(e.Graphics,
                            this.ClientRectangle);
  } else {
    this.Text = "Visual styles are disabled.";
    TextRenderer.DrawText(e.Graphics, this.Text,
         this.Font, new Point(0, 0), this.ForeColor);
  }
}
```

WPF

Principles of WPF

- Build a platform for rich presentation
- Build a programmable platform
- Build a declarative platform
- Integrate UI, documents, and media
- Incorporate the best of the Web, and the best of Windows
- Integrate developers and designers

WPF History

- 2001
  - A new team formed by Microsoft to build a unified presentation platform that could eventually replace User32/GDI32, Visual Basic, DHTML, and Windows Forms
- 2003
  - The Avalon project announced at Professional Developer Conference
- 2006
  - WPF released as a part of the .NET Framework 3.0
  - VS 2005 Extensions for .NET 3.0 (CTP)
- 2007
  - WPF included with Windows Vista
  - .NET Framework 3.5
  - Expression Blend 1.0
  - VS 2008 & VS WPF Designer
- 2008
  - WPF 3.5 SP1 (included in .NET 3.5 SP1)

Supported Systems

- WPF is included with:
  - Windows Vista
  - Windows Server 2008
- It is also available for:
  - Windows XP SP2
  - Windows Server 2003

WPF Features

- Graphical Services
  - All graphics are Direct3D applications
    - More advanced graphical features
    - Using Graphics Processing Unit of a graphics card
    - Vector-based graphics with lossless scaling
    - 3D model rendering
- Interoperability
  - WPF can be used inside Win32 code or WPF can use Win32 code
  - Windows Forms interoperability is possible using the **ElementHost** and **WindowsFormsHost** classes
- Annotations
  - WPF only provides the capability for creating, storing and managing annotations
  - Annotations can be applied on a per-object basis, for objects in a **Document** or **FlowDocument**

WPF Features cont.

- Media Services
  - 2D graphics with built-in set of brushes, pens, geometries, and transforms
  - 3D capabilities as a subset of the full feature Direct3D's set
  - Support for most common image formats
    - Support for Windows Imaging Component that allows to write image codecs
  - Support for WMF, MPEG and some AVI films
  - Support for Windows Media Player codecs
- Animations
  - Time-based animations
  - Animations can be triggered by other external events, including user action
  - Animation effects can be defined on a per-object basis
  - Set of predefined animation effects

WPF Features cont.

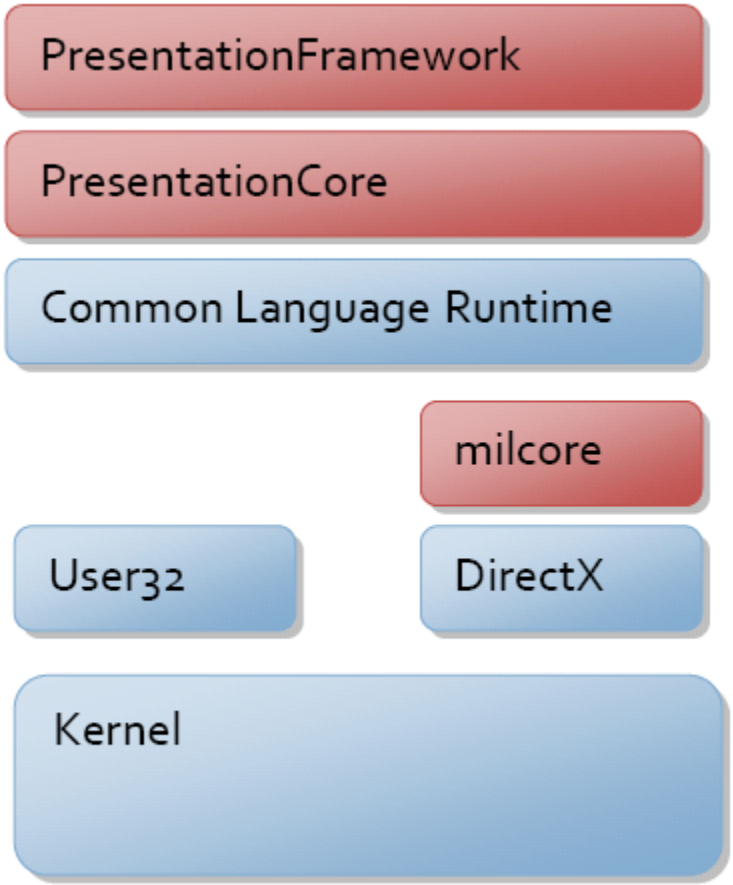
- Data binding
  - Three types of data binding:
    - One time: the client ignores updates on the server
    - One way: the client has read-only access to data
    - Two way: the client can read from and write data to the server
  - LINQ queries can act as data sources
- User interface
  - A set of built-in controls
  - A control's template can be overridden to completely change its visual appearance
  - Applications do not have to be bothered with repainting the display
- Documents
  - Support for XML Paper Specification documents
  - Supports reading and writing paginated documents using Open Packaging Convention

WPF Features cont.

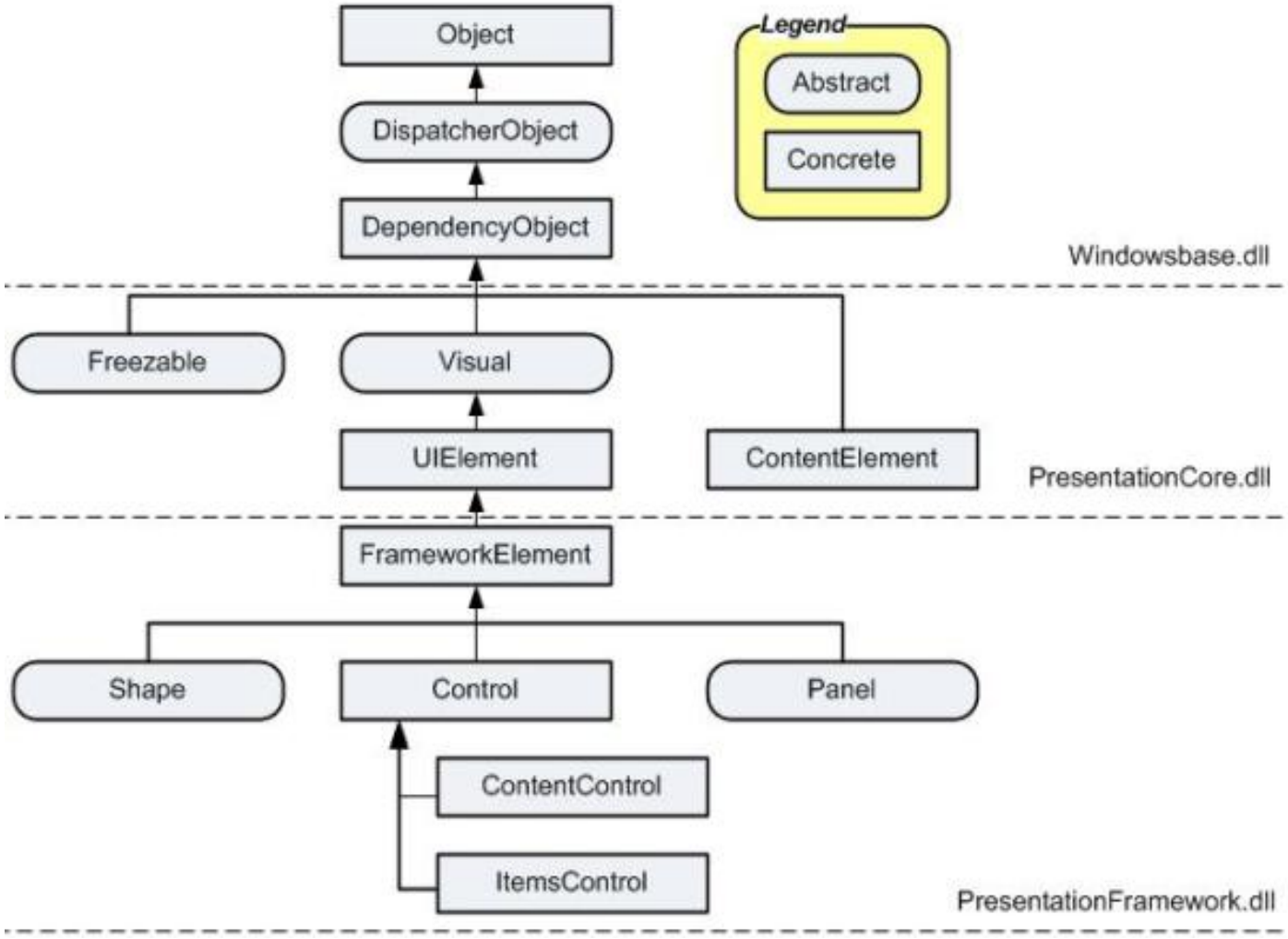
- Text
  - Support for OpenType, TrueType, and OpenType CFF fonts
  - WPF handles texts in Unicode
    - Independent of global settings, such as system locale
  - Built-in features: spell checking, automatic line spacing, enhanced international text, language-guided line breaking, hyphenation, justification, bitmap effects, transforms, and text effects such as shadows, blur, glow, rotation etc.
  - Support for animated text (both animated glyphs and real-time changes in postion, size, colour, and opacity)
- Accessibility
  - Microsoft UI Automation

Architecture

- PresentationFramework
  - End-user presentation features (including layouts, animations, and data-binding)
- PresentationCore
  - A managed wrapper for MIL
  - It implements the core services
- milcore – Media Integration Layer
  - It interfaces directly with DirectX
  - It is a native component



Fundamental Classes



http://windowsclient.net

System.Threading.DispatcherObject

- Most objects in WPF derive from **DispatcherObject**
- It provides the basic constructs for dealing with concurrency and threading
- WPF is based on a messaging system implemented by the dispatcher
  - The WPF dispatcher uses User32 messages for performing cross thread calls
- All WPF applications start with two threads: one for managing the UI and another background thread for handling rendering and repainting

System.Windows.DependencyObject

- One of the primary architectural philosophies used in building WPF was a preference for properties over methods or events
  - Properties are declarative and allow to more easily specify intent instead of action
- WPF provides a richer (than exists in CLR) property system, derived from the **DependencyObject** type
- Currently, the set of expressions supported is closed
- WPF properties support **change notifications**, which invoke bound behaviours whenever some property of some element is changed
  - Custom behaviors can be used to propagate a property change notification across a set of WPF objects

System.Windows.Media.Visual

- The **Visual** class provides for building a tree of visual objects, each optionally containing drawing instructions and metadata about how to render those instructions (clipping, transformation, etc.)
- It is the point of connection between these two subsystems, the managed API and the unmanaged **milcore**
  - WPF displays data by traversing the unmanaged data structures managed by the milcore
- The entire tree of visuals and drawing instructions is cached
  - WPF uses a retained rendering system
- Instead of clipping each component, each component is asked to render from the back to the front of the display
  - It allows to have complex, partially transparent shapes

System.Windows.UIElement

- **UIElement** defines core subsystems including Layout, Input, and Events
- Layout is a core concept in WPF
  - At the **UIElement** level, the basic contract for layout is introduced – a two phase model with Measure and Arrange passes
- Input originates as a signal on a kernel mode device driver
  - It gets routed to the correct process and thread through an intricate process involving the Windows kernel and User32
  - Once the User32 message corresponding to the input is routed to WPF, it is converted into a WPF raw input message and sent to the dispatcher
  - WPF allows for raw input events to be converted to multiple actual events

System.Windows.FrameworkElement

- It introduces a set of policies and customizations on the subsystems introduced in lower layers of WPF
  - The primary policy introduced by **FrameworkElement** is around application layout
- It also introduces a set of new subsystems
  - The data binding subsystem allows to bind properties to a piece of data
    - WPF has full support for property binding, transformation, and list binding
    - Data templates allow you to declaratively specify how a piece of data should be visualized
  - Styling is really a lightweight form of data binding
    - It allows to bind a set of properties from a shared definition to one or more instances of an element

System.Windows.Controls.Control

- **Control**'s most significant feature is templating
  - Templating allows a control to describe it's rendering in a parameterized, declarative manner
- A common aspect of the data model of controls is the content model
  - E.g., content for a button can either be a simple string, a complex data object, or an entire element tree
  - In the case of a data object, the data template is used to construct a display

XAML

- Extensible Application Markup Language (XAML) is a markup language for declarative application programming
  - The developer (or designer) describes the behaviour and integration of components without the use of procedural programming
- Using XAML to develop user interfaces allows for separation of model and view
  - However, all elements of WPF may be coded e.g. in C#
- The XAML code can ultimately be compiled into a managed assembly in the same way all .NET languages are
- XAML is not specific to WPF (or even .NET), however, it has been introduced as integral part of WPF

XAML Examples

```
<Application x:Class="WpfApp.App"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    StartupUri="Window1.xaml">
     <Application.Resources>

     </Application.Resources>
</Application>
```

```
<Window x:Class="WpfApp.Window1"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="Window1" Height="300" Width="300">
     <Grid>
     </Grid>
</Window>
```

XAML Namespaces

- XML namespaces are declared using attributes
  - These attributes can be placed inside any element start tag, but usually they are declared in the very first tag
  - Once a namespace is declared, it can be used anywhere in the document
- There are two basic namespaces:
  - **http://schemas.microsoft.com/winfx/2006/xaml**
    - the XAML namespace which includes various XAML utility features
    - by default, this namespace is mapped to the prefix **x**, so it can be applied by placing the namespace prefix before the element name (`<x:ElementName>`)
  - **http://schemas.microsoft.com/winfx/2006/xaml/presentation**
    - the core WPF namespace which encompasses all the WPF classes including all controls
    - by default, it is declared without a namespace prefix, so it becomes the default namespace for the entire document

Application Class

- The **Application** object is responsible for managing the lifetime of the application, tracking the visible windows, dispensing resources, and managing the global state of the application
- A WPF application logically starts executing when the **Run** method is invoked on an instance of the Application object

```
using System;
using System.Windows;

namespace WpfApplication1 {
  static class Program {
    [STAThread]
    static void Main() {
      Application app = new Application();
      Window w = new Window();
      w.Title = "Hello World";
      w.Show();
      app.Run();
    }
  }
}
```

Application's Lifetime

1. **Application** object is constructed
2. **Run** method is called
3. **Application.Startup** event is raised
   - Using the Startup event is a preferred place for application initialization (as opposed for the constructor)
4. User code constructs one or more **Window** objects
5. **Application.Shutdown** method is called
6. **Application.Exit** event is raised
7. **Run** method completes

Error Handling

- The **Application.DispatcherUnhandledException** event is raised when the dispatcher sees an unhandled exception
- The **DispatcherUnhandledExceptionEventArgs.Handled** flag indicates if the exception should be ignored and the application should continue to run

```
<Application [...]
     DispatcherUnhandledException="App_UnhandledException">
</Application>
```

```
public partial class App : Application
{
    private void App_UnhandledException(object sender,
            DispatcherUnhandledExceptionEventArgs e) {
        using (StreamWriter errorLog =
                new StreamWriter("c:\\error.log", true)) {
            errorLog.WriteLine("Error @ " + DateTime.Now.ToString());
            errorLog.WriteLine(e.Exception.ToString());
        }
        e.Handled = true;
    }
}
```

Application's State

- The application object is available globally using the **Application.Current** static property
- The **Application.Properties** is a dictionary of any custom data stored at the level of the application

```
Application.Current.Properties["LastError"] = e.Exception;
```

```
object lastError = Application.Current.Properties["LastError"];
if (lastError != null &&
    lastError is DivideByZeroException) {
}
```