

Family.Show – A WPF Reference Application

Ralph Arvesen, Alan Le, Paul Osburn and Tony Sokolowski
Vertigo

July 2007

Contents

Family.Show – A WPF Reference Application	1
Overview	1
Tools and process	2
Project Files	4
Architecture	6
Resources	8
Diagram	10
ListView	15
Common file dialogs	22
Data model	24
GEDCOM classes	33
ClickOnce	33
Extending Family.Show	33
More information	34

Overview

Family.Show is a Windows Presentation Foundation (WPF) reference application. Vertigo Software collaborated with Microsoft to build Family.Show as a tool for developers to use as a reference for building their own applications using WPF. In addition, Family.Show is intended to be a functional application that genealogy enthusiasts and everyday users can use to build their family trees and family histories.

The goal of this whitepaper is to discuss in detail how Family.Show was created.

Key Features

Some of the key features of Family.Show are:

- Quickly build your family tree
- Add photos easily via drag-and-drop
- Create advanced family tree visualizations with pan and zoom support
- See what your family tree looked like years ago using the Time Explorer
- Tell your family members' story using rich editing and formatting controls
- Mine your family data with statistical filtering and sorting
- Import and export family trees to GEDCOM 5.5, a standard genealogy format
- Change the look of the entire application by choosing a different skin

To create the application, we took advantage of some of the powerful features built into WPF:

- Two-way Data Binding
- Flow Document Viewer
- Rich Text Editor
- Animations
- Styles, Resources, and Templates
- XML Serialization
- Custom Controls
- Command Pattern
- ClickOnce Deployment

Family.Show is not a performance sample and works best with smaller sets of data. The application is available at <http://www.vertigo.com/familyshow>.

Requirements

Family.Show runs on Microsoft Windows Vista and Microsoft Windows XP. Microsoft Windows Vista includes the .NET Framework 3.0 making it easy to run Family.Show. To run the application on Microsoft Windows XP, you must first download and install version 3.0 of the .NET Framework. The framework can be downloaded at www.netfx3.com.

Tools and process

Several tools were used to create Family.Show, including Microsoft Visual Studio 2005, Microsoft Expression Blend, and Microsoft Expression Design. Though the project team was small (4 people), we used Microsoft Team System for source code control. The ability to work on the same files at the same time and use Microsoft Team System to merge changes worked very well and helped speed up development.

The Family.Show user interface was built with Expression Blend to create the layout, adjust the look-and-feel, set properties, set data bindings and create animations. The Visual Studio XAML editor was used to make small adjustments to the XAML files.

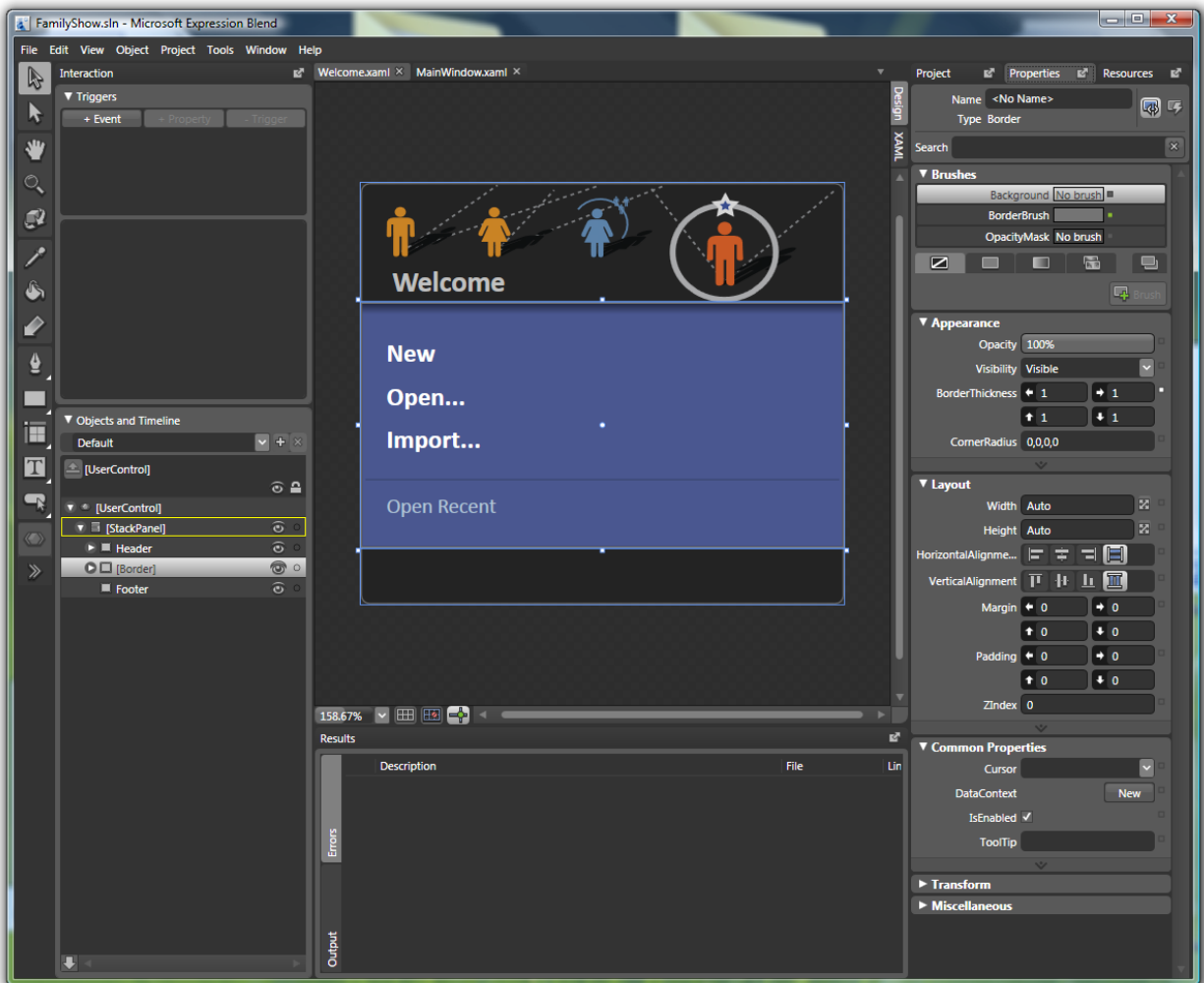


Figure 1 - Microsoft Expression Blend

WPF and the Microsoft Expression tools enabled close collaboration between the designer and developers on the project. The following was the process using these tools:

- Designer creates design assets using Expression Design or other graphic design tools
- Designer or developer exports asset to XAML using Expression Design
- Developer uses the exported XAML in the application, adding it to the project using either Expression Blend or the XAML editor in Visual Studio

When we first started development, XAML was created by hand using the XAML editor in Visual Studio. Over time the developers learned how to use Expression Blend which, in many cases, significantly reduced development time. This was especially true when making final adjustments to the visuals

before a release. Editing the XAML, rebuilding in Visual Studio, and then running the application to view the changes (and doing this over and over again) to make adjustments took too much time. Making the change in Expression Blend and seeing it immediately turned out to be a big savings in development time.

Debugging

In addition to the debugging features built into Visual Studio, Snoop (<http://www.blois.us/Snoop>) is a great tool to explore WPF applications; it displays the visual tree as well as properties and events. It can be used to map UI elements to classes in the Family.Show project.

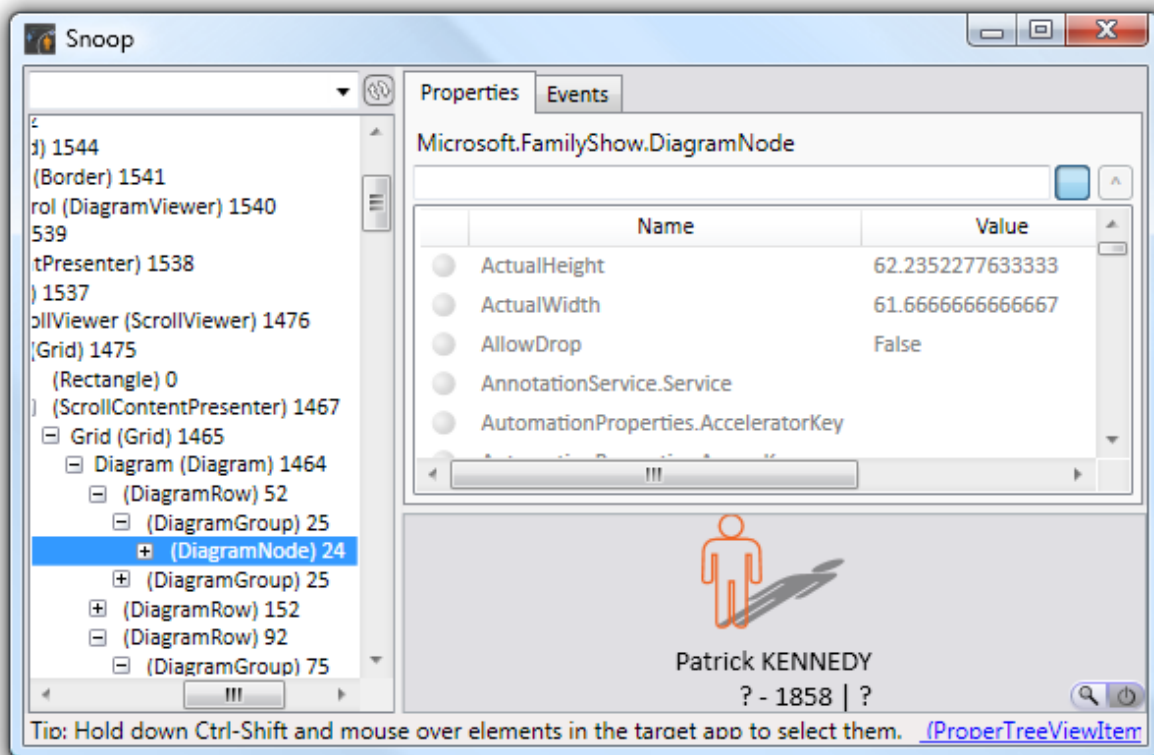


Figure 2 - Using Snoop to explore WPF applications.

Project Files

The solution is separated into two projects. **FamilyShow** is a .NET Framework 3.0 WPF Windows Application containing the user interface and related classes. **FamilyShowLib** is a .NET Framework 3.0 class library containing the data object and GEDCOM classes referenced by the **FamilyShow** project. All source code in code-behind files is written in C#.

The following is an overview of the files in each project:

FamilyShow Project Files

App	WPF application class contains data common to all classes.
CommonDialogs	Displays the new Vista-style open and save dialogs.
MainWindow	Window for the application containing all of the other controls.
ValueConverters	Value converters used in all of the XAML pages.

Controls

Details	The right pane in the diagram view.
NewUserControl	Control that is displayed when a new file is created.
Welcome	Control that is displayed when the application first runs.

Controls\Diagram

Diagram	Displays a diagram of the family members.
DiagramConnector	Draws connector lines between two diagram nodes.
DiagramGroup	Contains and arranges one or more DiagramNode objects.
DiagramLogic	Logic that determines which nodes appear in the diagram.
DiagramNode	A person in the diagram.
DiagramRow	Contains and arranges one or more DiagramGroup objects.
DiagramViewer	Wrapper around the diagram that supports zooming and panning.

Controls\FamilyData

FamilyData	View that is displayed when the Expand button is clicked. Contains the editable ListView and chart controls.
FamilyDisplayListView	ListView control in the diagram view.
FamilyEditListView	ListView control in the family data view.
FilterSortListView	Base class that filters a ListView control. Also sorts the list since it derives from SortListView .
FilterText	UserControl with a text box and reset button.
Histogram	Chart that displays age ranges.
SharedBirthdays	List that groups people by birthday.
SortListView	Base class that sorts a ListView control.
TagCloud	List that displays frequency of last name.

Images

*.png	Images used for the RichTextBox editor toolbar buttons.
-------	--

Sample Files

Kennedy.ged, Windsor.family	Sample GEDCOM and Family.Show files that are installed when the application first runs.
--------------------------------	---

Skins

*.xaml	Resources specific to the Default and Gray skins.
--------	---

FamilyShowLib Project Files

GedcomConverter,	Import and export to GEDCOM.
------------------	------------------------------

GedcomExport,
GedcomImport

People	Collection of Person objects.
Person	Person in the People list.
Photo	Photo for a person.
Relationships	Relationships between Person objects.
RelationshipsHelper	Relationship logic rules for adding people and how they relate to each other.
Story	Story for a person.

Table 1-Solution Files

Project Dependencies

In addition to the Family.Show source code and Visual Studio, you'll also need the Microsoft Visual Studio 2005 CTP Extensions for .NET Framework 3.0 to compile the application. The extensions can be found at www.netfx3.com.

Architecture

The Family.Show application architecture is best described by breaking it down into its major user interface features and by detailing the underlying data model.

Main window

The **MainWindow** class serves as the main entry to the Family.Show application. In fact, it is the only WPF Window in the project. It is the container for user controls such as the **Diagram** and **Details** controls and is in charge of the commands (e.g. menu items) and event handlers.



Figure 3 - Main Window Layout

The main window layout is comprised of a **DockPanel** containing three container controls: **HeaderBorder**, **MenuBar**, and **MainGrid**. A **DockPanel** is used since it makes it easy to dock the header and menu on the top of the window. The **MainGrid** contains the **DiagramPane**, **DetailsPane**, **NewUserController**, **WelcomeUserController**, **PersonInfoControl**, and **FamilyDataControl**. It is a **Grid** control to allow the diagram and details sections to be collapsible and expandable.

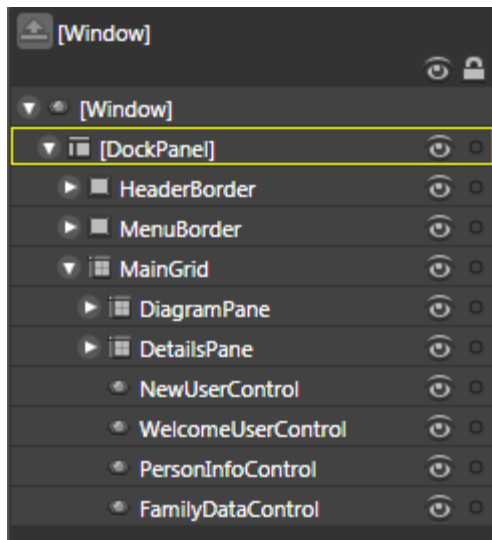


Figure 4 - Main Window Objects

Routed events

The main window controls the display of the user controls that it contains. For example, when the window is loaded, it shows the welcome screen. Based on the actions that the user takes for the welcome screen, which are exposed as routed events, the main window shows or hides the diagram and details controls.

```
<!-- Welcome User Control -->
<local:Welcome x:Name="WelcomeUserControl"
    HorizontalAlignment="Center"
    VerticalAlignment="Center"
    ImportButtonClick="WelcomeUserControl_ImportButtonClick"
    NewButtonClick="WelcomeUserControl_NewButtonClick"
    OpenButtonClick="WelcomeUserControl_OpenButtonClick"
    OpenRecentFileButtonClick="WelcomeUserControl_OpenRecentFileButtonClick" />
```

Listing 1 – Welcome User Control Event Handler

Resources

WPF resources provide a simple way to reuse styles, brushes, and other formatting objects in multiple places in an application. This allows a consistent look and behavior across the application and makes formatting and styling easier to maintain since changes can be made in fewer places. This is analogous to the Cascading Style Sheets (CSS) used by Web applications.

Declaring resources in XAML instead of creating them programmatically makes resources easier to modify and allows designers to work directly with the application instead of using mock-ups. All resources in Family.Show are declared in XAML.

Resource Organization

Since Family.Show uses a fairly large number of resources, we separated them across multiple files organized by functional group to make them easier to find. This also helped to reduce duplication by more easily enabling resource sharing across controls.

The resources in Family.Show are in the **FamilyShow** project and are organized as shown below. There are two skins that the application can use: a Default skin and a Gray skin.

FamilyShow Project

Skins

Default

DefaultResources.xaml

Resources

BrushResources.xaml

ControlResources.xaml

ConverterResources.xaml

DataTemplates.xaml

DiagramResources.xaml

GlobalResources.xaml

SimpleStyles.xaml

Gray

GrayResources.xaml

Resources

...

Merging Resources

Family.Show contains application-specific resources and skin-specific resources. Application resources are common for all skins and skin resources are specific to each skin (Default and Gray). The skin resources are grouped into different XAML files and specified in the **MergedDictionaries** collection. The result is one global resource dictionary that contains application and skin resources and is available to all application elements.

```
<ResourceDictionary xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
    <ResourceDictionary.MergedDictionaries>
        <ResourceDictionary Source="Resources\SimpleStyles.xaml"/>
        <ResourceDictionary Source="Resources\BrushResources.xaml"/>
        <ResourceDictionary Source="Resources\ControlResources.xaml"/>
        <ResourceDictionary Source="Resources\ConverterResources.xaml"/>
        <ResourceDictionary Source="Resources\DataTemplates.xaml"/>
```

```

    <ResourceDictionary Source="Resources\DiagramResources.xaml"/>
    <ResourceDictionary Source="Resources\GlobalResources.xaml"/>
  </ResourceDictionary.MergedDictionaries>
</ResourceDictionary>

```

Listing 2 - Merging Resources

Diagram

The **Diagram** class is responsible for displaying the immediate family which consists of the primary person, current spouses, previous spouses, full siblings, half siblings, parents and children.

Diagram Layout

One option for this control is to arrange all of the nodes on a single **Canvas**. Instead, the application uses a series of rows to simplify the layout process. Each row contains one or more groups, and each group contains one or more nodes (a person). This makes it easy to lay out the diagram since each component only arranges its content. In Family.Show, rows, stacked vertically, contain groups, stacked horizontally, which contain nodes. You can see this by running a debug build and selecting **Show Diagram Outline** from the context menu; the gold lines show the rows and gray lines show the groups.

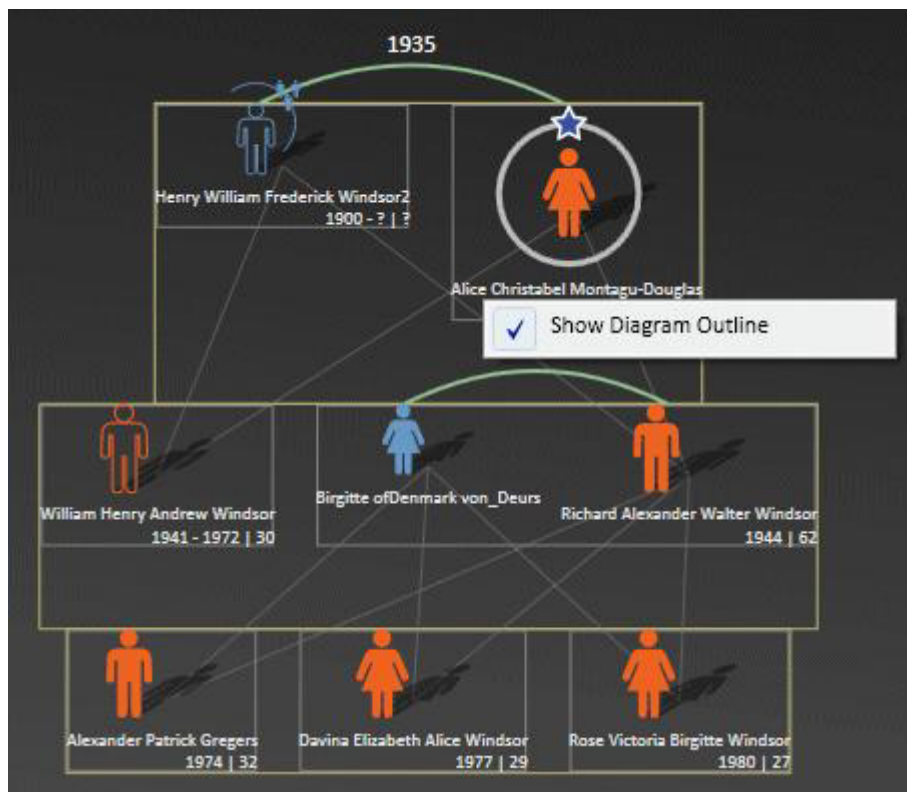


Figure 5 - Diagram Control with rows, groups, and nodes

WPF has a two-phase layout system where it first measures controls and then arranges the controls. Controls participate in this layout system by overriding the methods **MeasureOverride**, **ArrangeOverride**, and **GetVisualChild** and the property **VisualChildrenCount**.

MeasureOverride is called when the control should calculate its desired size. It does this by asking all of its children to calculate their size. The following code snippet shows how the **DiagramRow** class overrides this method and measures all of the child group controls.

```
protected override Size MeasureOverride(Size availableSize)
{
    // Let each group determine how large they want to be.
    Size size = new Size(double.PositiveInfinity, double.PositiveInfinity);
    foreach (DiagramGroup group in groups)
        group.Measure(size);

    // Return the total size of the row.
    ...
}
```

Listing 3 - MeasureOverride

The **ArrangeOverride** method is called when the control should arrange its children. The following code shows how the **DiagramRow** overrides this method and calls the **Arrange** method on each child group control.

```
protected override Size ArrangeOverride(Size finalSize)
{
    // Arrange the groups in the row, return the total size.
    return ArrangeGroups(true);
}

private Size ArrangeGroups(bool arrange)
{
    // Position of the next group.
    double pos = 0;

    // Bounding area of the group.
    Rect bounds = new Rect();

    // Total size of the row.
    Size totalSize = new Size(0, 0);

    foreach (DiagramGroup group in groups)
    {
        // Group location.
        bounds.X = pos;
        bounds.Y = 0;
    }
}
```

```

        // Group size.
        bounds.Width = group.DesiredSize.Width;
        bounds.Height = group.DesiredSize.Height;

        // Arrange the group, save the location.
        if (arrange)
        {
            group.Arrange(bounds);
            group.Location = bounds.TopLeft;
        }

        // Update the size of the row.
        totalSize.Width = pos + group.DesiredSize.Width;
        totalSize.Height = Math.Max(totalSize.Height, group.DesiredSize.Height);

        pos += (bounds.Width + this.groupSpace);
    }

    return totalSize;
}

```

Listing 4 - Layout of Diagram Nodes

The **VisualChildrenCount** property is called to get the number of child controls and the **GetVisualChild** method is called to get a specific child control.

```

protected override int VisualChildrenCount
{
    // Return the number of groups.
    get { return groups.Count; }
}
protected override Visual GetVisualChild(int index)
{
    // Return the requested group.
    return groups[index];
}

```

Listing 5 - Working with Diagram Child Nodes

The diagram is implemented in the **Diagram** class, rows in the **DiagramRow** class, and groups in the **DiagramGroup** class. The class **DiagramLogic** contains business logic that determines which nodes are displayed in the diagram.

Diagram nodes

A diagram node represents one person in the diagram and is implemented in the **DiagramNode** class. It derives from **Button** so the **Click** event can be handled. Each node is bound to a **Person** object. The nodes appearance is determined by the state of the **Person** object; for example, if the person is male or

female, deceased or living, or a spouse or child. All visual aspects of a node are declared in XAML in the **DiagramResources.xaml** file under the **Skins** folder.

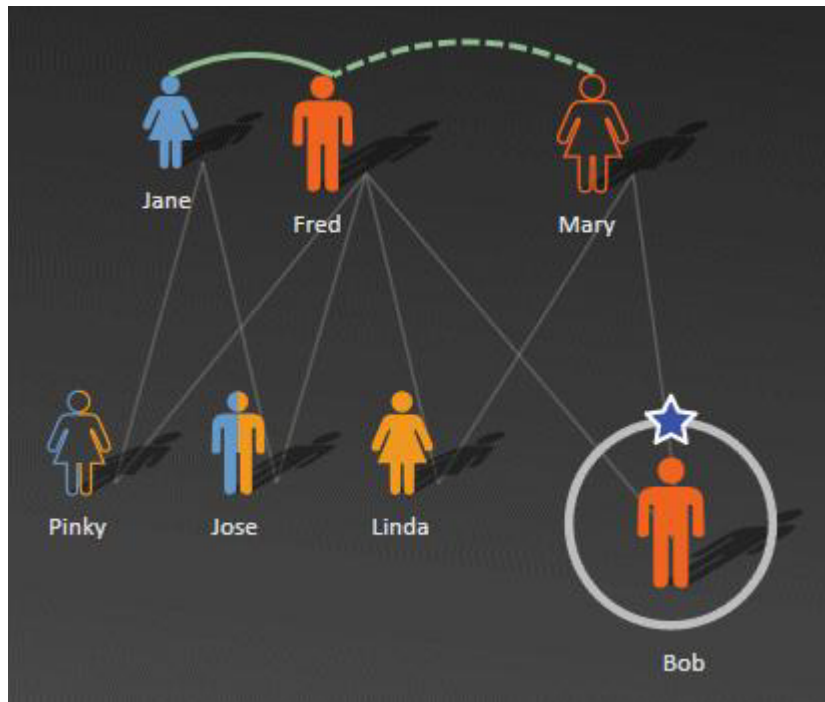


Figure 6 - Different Types of Nodes

Nodes are scaled as they increase in generations but only the drawing element of the node is scaled, not the textual information below the node. This is accomplished by overriding **OnApplyTemplate** and finding the drawing element within the template, and then only scaling that part of the template.

```
public override void OnApplyTemplate()
{
    // Scale the person drawing part of the node, not the entire node.
    FrameworkElement personElement = this.Template.FindName(
        "Person", this) as FrameworkElement;

    if (personElement != null)
    {
        ScaleTransform transform = new ScaleTransform(this.scale, this.scale);
        personElement.LayoutTransform = transform;
    }

    base.OnApplyTemplate();
}
```

Listing 6 - Scaling Diagram Nodes

Diagram connectors

Diagram connectors are lines drawn between two nodes that represent child, parent, or spouse relationships.

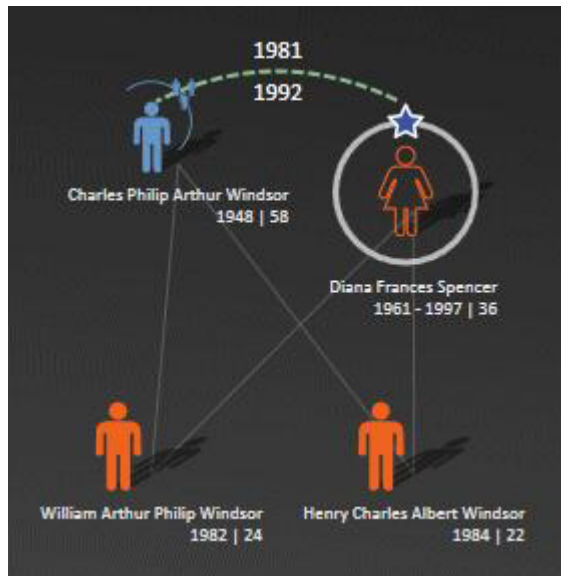


Figure 7 - Different Diagram Connectors

The connectors could have been implemented as visual elements and added to the visual tree of the diagram. Instead, the connectors are drawn at a lower level to save resources in the WPF layout system. The following code shows how the diagram class overrides **OnRender** and draws each connector.

```
/// <summary>
/// Draw the connector lines at a lower level (OnRender) instead
/// of creating visual tree objects.
/// </summary>
protected override void OnRender(DrawingContext drawingContext)
{
    // Draw child connectors first, so marriage information appears on top.
    foreach (DiagramConnector connector in logic.Connections)
    {
        if (connector.IsChildConnector)
            connector.Draw(drawingContext);
    }
}
```

```

// Draw all other non-child connectors.
foreach (DiagramConnector connector in logic.Connections)
{
    if (!connector.IsChildConnector)
        connector.Draw(drawingContext);
}
}

```

Listing 7 - Drawing Connectors

The **DiagramConnector** class is the base class of the **ChildDiagramConnector** and **MarriedDiagramConnector** classes. The derived classes use the **DrawingContext** class to draw lines between two nodes or arcs with optional text above and below the arc. The following shows how a line is drawn between two nodes.

```

override public bool Draw(DrawingContext drawingContext)
{
    drawingContext.DrawLine(this.Pen,
        this.StartNode.Center, this.EndNode.Center);

    return true;
}

```

Listing 8 - Drawing a Line between Two Nodes

Animations are still supported even though the drawing is performed at a lower level. For example, the pen that is used to draw the connector line can be animated as shown below.

```

DoubleAnimation animation = new DoubleAnimation();
animation.From = 0.0;
animation.To = 1.0;
animation.Duration = 300;
connectorPen.Brush.BeginAnimation(Brush.OpacityProperty, animation);

```

Listing 9 - Animations

Diagram viewer

The diagram is contained in the **DiagramViewer** class that is responsible for zooming in and out and panning. Zooming is accomplished by using a **ScaleTransform** and panning with the **ScrollViewer** class.

List View

Family.Show uses two ListViews to display the list of people. The first is on the diagram view and supports sorting and filtering. The second is on the family data view and supports sorting, filtering, and editing.

Name	Born	Died	Age	
Beatrice Elizabeth Man	1988	-	18	
Benjamin	-	-		
Bessiewallis Warfield	-	1986		
Birgitte of_Denmark vo	-	-		
Charles Philip Arthur W	1948	-	58	
David Lascelles	-	-		
David Albert Charles Ar	1961	-	45	

Figure 8 - List View Control in Main Window

First Name	Last Name	Age	Birth Date	Birth Place	Death Date
Paul	Mowatt				
Angus	Ogilvy				
Marina Victoria /	Ogilvy	40	7/31/1966	Thatched House,L	
James Robert Br	Ogilvy	43	2/29/1964	Thatched House,L	
Mark Anthony P	Phillips	58	9/22/1948		
Zara Anne Elizab	Phillips	26	5/15/1981	St. Marys Hosp.,P	
Peter Mark Andr	Phillips	29	11/15/1977	St. Mary's Hosp.,P	
Julia	Rawlinson				
Diana Frances	Spencer	36	7/1/1961	Park House,Sandr	8/31/1997
Marion (Maria) E	Stein				

Figure 9 - List View Control in Family Data View

ListView sorting

The base class `SortListView` implements a sortable `ListView` control. Sorting the data is accomplished by adding a **SortDescription** object to the view as shown below.

```
private void SortList(string propertyName)
{
    // Get the data to sort.
    ICollectionView dataView = onViewSource.DefaultView(this.ItemsSource);
```



```

    // Specify the new sorting information.
    dataView.SortDescriptions.Clear();
    SortDescription description = new SortDescription(propertyName, action);
    dataView.SortDescriptions.Add(description);

    dataView.Refresh();
}

```

Listing 10 - Sortable ListView Control

The more challenging aspect is getting the name of the property that is used to sort the data when the column header is clicked. Most samples use a simple idea, such as the column header text, but in real-world applications the header text will probably be different from the underlying property name. For example, here is the XAML from the MSDN article that shows how to sort a ListView.

```

<GridView>
    <GridViewColumn DisplayMemberBinding="{Binding Path=Year}"
        Header="Year" Width="100"/>
</GridView>

```

Listing 11 - Sorting, Example 1

Notice that the **DisplayMemberBinding** is bound to a property with exactly the same name as the **Header** text, but this does not work if you want to bind to a different property such as BirthYear instead of Year.

You can get the sort property from **DisplayMemberBinding** instead of the **Header** text, but that creates a new problem; **DisplayMemberBinding** takes precedence over **CellTemplate** so now you lose the ability to style the cell. For example, **CellTemplate** is ignored in the following XAML.

```

<GridView>
    <GridViewColumn DisplayMemberBinding="{Binding Path=Year}"
        CellTemplate="{StaticResource YearColumnTemplate}"
        Header="Year" Width="100"/>
</GridView>

```

Listing 12 - Sorting, Example 2

Family.Show solves this problem by creating the **SortListViewColumn** class. This is a simple class that derives from **GridViewColumn** and contains two dependency properties called **SortProperty**, the name of the property used to sort the data, and **SortStyle**, the style to apply when the column is sorted. The following code shows how the **SortListViewColumn** class is used to specify the **Header** text and **CellTemplate**, as well as the **SortProperty** and **SortStyle**.

```

<GridView>
    <local:SortListViewColumn Header="Birth Date" Width="80"
        SortProperty="BirthDate" SortStyle="FamilyDataGridViewColumnHeader"
        HeaderContainerStyle="{DynamicResource FamilyDataGridViewColumnHeader}"

```

```

        CellTemplate="{StaticResource BirthDateColumnTemplate}" />
</GridView>

```

Listing 13 - Sorting in Family.Show

Now it's trivial for the code to get the property that is used to sort the data.

```

private void OnHeaderClicked(object sender, RoutedEventArgs e)
{
    // Get the ListView column for the header that was clicked.
    GridViewColumnHeader header = e.OriginalSource as GridViewColumnHeader;
    SortListViewColumn column = header.Column as SortListViewColumn;

    ...

    // Sort the data.
    SortList(column.SortProperty);

    ...
}

```

Listing 14 - Sorting when Header is Clicked

ListView filtering

The class **FilterSortListView** derives from **SortListView** and adds filtering capabilities to the **ListView** control. This is accomplished by setting the **Filter** property of the view. For example:

```

/// <summary>
/// Filter the list.
/// </summary>
private void FilterList(string text)
{
    // Get the data the ListView is bound to.
    ICollectionView view = CollectionViewSource.GetDefaultView(this.ItemsSource);

    ...

    // Filter the list.
    view.Filter = new Predicate<object>(FilterCallback);
}

/// <summary>
/// This is called for each item in the list.
/// </summary>
virtual protected bool FilterCallback(object item)
{
    // Return true if the item should be in the list.
    ...
}

```

```
}
```

Listing 15 - Filtering a ListView

Filtering can occur on the UI thread, during idle time, or a background thread. The Family.Show application performs filtering during idle time using the **Dispatcher** object. This is an easy way to execute code in the background, but not worry about threading issues.

```
/// <summary>
/// Filter the data using the specified filter text.
/// </summary>
public void FilterList(string text)
{
    ...

    // Start an async operation that filters the list.
    this.Dispatcher.BeginInvoke(
        DispatcherPriority.ApplicationIdle,
        new FilterDelegate(FilterWorker));
}
```

Listing 16 - Filtering on a Background Thread

This allows the ListView control to filter the result without blocking the UI thread from updating. This provides a better user experience since there is less of a delay as each character is entered in the filter text box.

Derived classes

Since sorting and filtering are handled in base classes, the two instances of the ListView controls are actually very simple.

FamilyDisplayListView and **FamilyEditListView** derive from **FilterSortListView** and contain only one method that overrides **FilterCallback** that is called when the data needs to be filtered. The entire **FamilyDisplayListView** is shown below.

```
class FamilyDisplayListView : FilterSortListView
{
    /// <summary>
    /// Called for each item in the list. Return true if the item should be in
    /// the current result set, otherwise return false to exclude the item.
    /// </summary>
    protected override bool FilterCallback(object item)
    {
        Person person = item as Person;
        if (person == null)
            return false;

        if (this.Filter.Matches(person.Name) ||
```

```

        this.Filter.MatchesYear(person.BirthDate) ||
        this.Filter.MatchesYear(person.DeathDate) ||
        this.Filter.Matches(person.Age))
        return true;

    return false;
}
}

```

Listing 17 - FilterCallback

Filter control

The **FilterText** class is a **UserControl** that provides the filter user interface. It consists of a **Border**, **TextBox** and reset **Button**. The reset button is hidden and displayed based on the content of the TextBox.

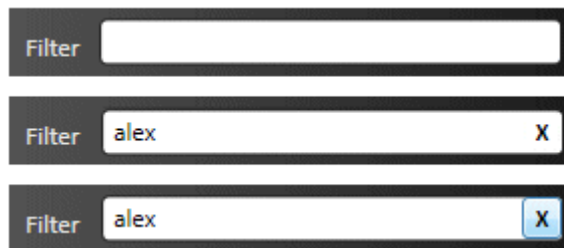


Figure 10 - Different States of Filter Control

ListView editing

Developers have been writing code to make ListView controls editable ever since the control became part of the Windows operating system. In the past, this was accomplished by showing and hiding a text box at the correct location when a cell received and lost focus. This was not a lot of code, but it's not trivial and had its own set of challenges such as deciding when a cell receives and loses focus. This is the approach the SDK sample takes which you can see at <http://msdn2.microsoft.com/en-us/library/ms771277.aspx> and http://blogs.msdn.com/atc_avalon_team/archive/2006/03/14/550934.aspx.

The Family.Show application demonstrates a different technique by declaring everything in XAML and avoids creating a custom text box that is shown and hidden. The **CellTemplate** specifies the template that is used for the cell.

```

<local:SortListViewColumn Header="First Name" Width="100"
    SortProperty="FirstName" SortStyle="FamilyDataGridViewChildHeader"
    HeaderContainerStyle="{DynamicResource FamilyDataGridViewChildHeader}"
    CellTemplate="{StaticResource FirstNameColumnTemplate}" />

```

Listing 18 - In-line Editing using CellTemplate

The template contains the UI elements and binding information. In this case, a `TextBox` control is bound to the **FirstName** property.


```
<DataTemplate x:Key="FirstNameColumnTemplate">
    <TextBox Style="{StaticResource TextBoxStyle}" Text="{Binding Path=FirstName}" />
</DataTemplate>
```

Listing 19 - Editing Data Template

The **TextBoxStyle** contains a trigger on **IsFocused** which toggles the cell into edit mode. The cell is always in edit mode but its visual state changes to indicate its content can be modified.

```
<Style x:Key="TextBoxStyle" TargetType="{x:Type TextBox}" >
    <Setter Property="Background" Value="#00ffffff" />
    <Setter Property="BorderBrush" Value="#00ffffff" />
    <Setter Property="AcceptsReturn" Value="False" />
    <Setter Property="AcceptsTab" Value="False" />
    <Setter Property="Foreground" Value="{DynamicResource FamilyDataFontColor}" />
    <Style.Triggers>
        <Trigger Property="IsFocused" Value="True">
            <Setter Property="Foreground" Value="#ff000000" />
            <Setter Property="BorderBrush" Value="#ff000000" />
            <Setter Property="Background" Value="#a0ffffff" />
        </Trigger>
    </Style.Triggers>
</Style>
```

Listing 20 - TextBoxStyle



Mark Anthony Phillips	58
Zara Anne Elizabeth Phillips	26
Peter Mark Anderson Phillips	29

Figure 11 - Editing Cells in List View Control

Now we have two-way binding to the underlying data and the change between display mode and edit mode is handled in XAML.

List View performance

The **ListView** control can work with a virtual list or non-virtual list. A virtual list generates UI elements for only visible cells whereas a non-virtual list generates UI elements for all cells. Using a virtual list makes a huge difference in performance and is set with the **VirtualizingStackPanel.IsVirtualizing** property. You have to be careful since other properties, such as **ScrollViewer.CanContentScroll**, turn off virtualizing.

```
<local:FamilyEditListView x:Name="FamilyEditor"
```

```
ScrollView.HorizontalScrollBarVisibility="Disabled"  
ScrollView.VerticalScrollBarVisibility="Auto"  
VirtualizingStackPanel.IsVirtualizing="True"  
ScrollView.CanContentScroll="True" >
```

Listing 21 - Virtualizing StackPanel

The performance difference depends on several factors such as size of the list, number of columns, and the elements in the cell templates. The following are the results of a test we ran on the **FamilyEditListView** class.

	Virtual List	Non-virtual List
Sort column	2 seconds	66 seconds
Filter list	2 seconds	142 seconds
Working set	51K	221K

Table 2-ListView Performance

Common file dialogs

Family.Show displays common file dialogs when opening and savings files. Unfortunately the .NET **OpenFileDialog** and **SaveFileDialog** classes display the old-style file dialogs since they pass a hook to the underlying API functions.

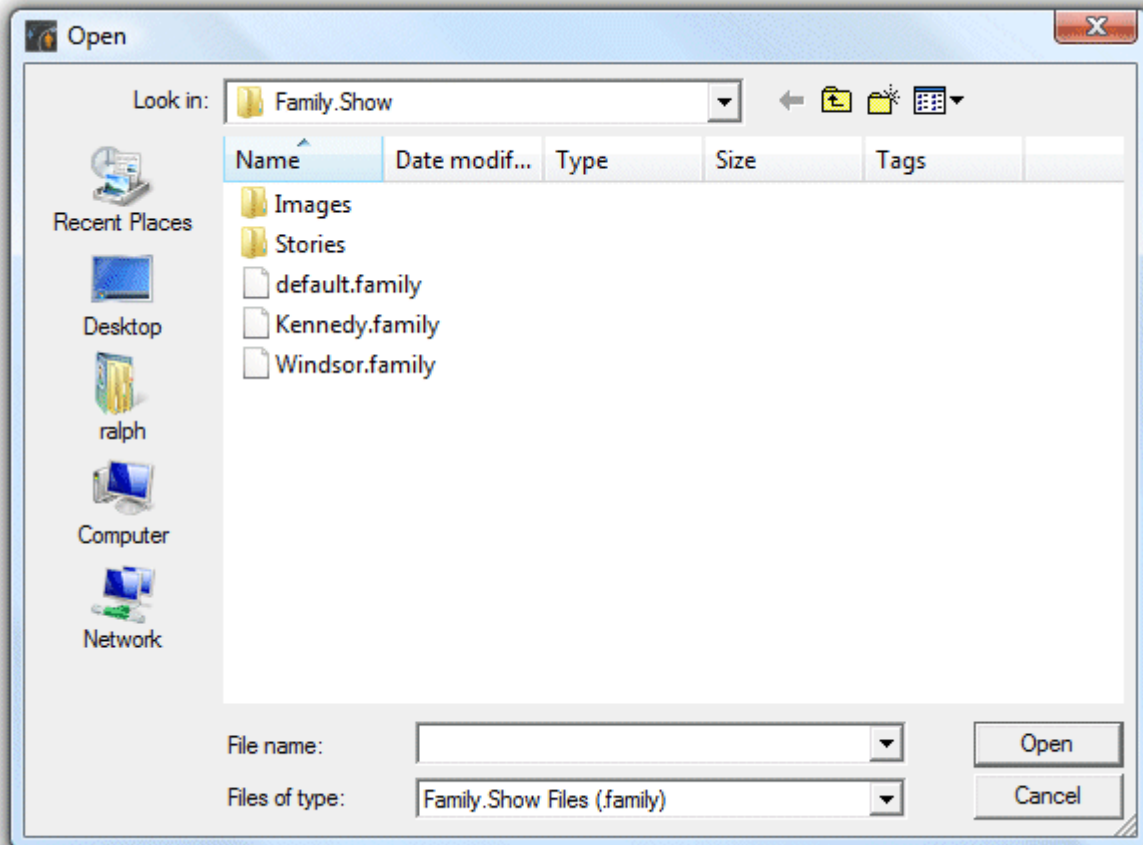


Figure 12 - Old-Style Common Dialogs

The Family.Show **CommonDialog** class displays the new Vista-style dialogs by calling the **GetOpenFileName** and **GetSaveFileName** API functions directly and not passing a hook.

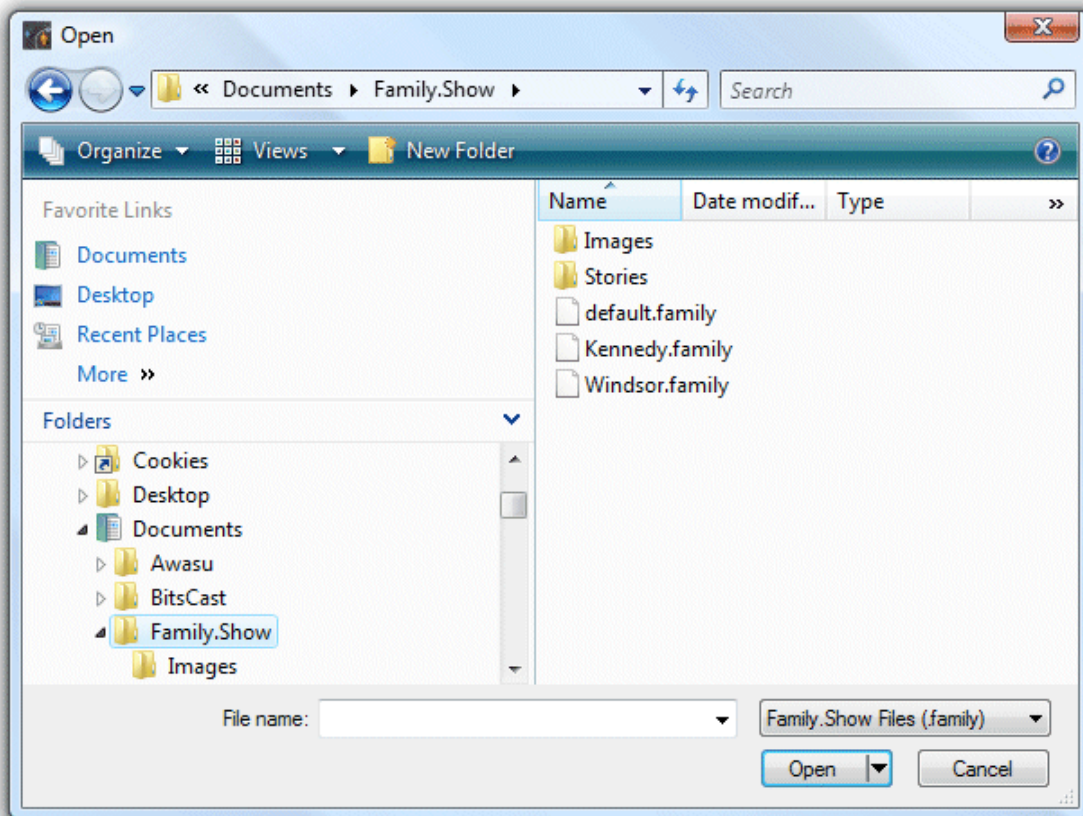


Figure 13 - Displaying the New, Vista-style Common Dialog

A hook is only required when you want to customize the common file dialogs. If a hook is required for your application, see the Vista Bridge sample at <http://msdn2.microsoft.com/en-us/library/ms756482.aspx> that contains managed wrappers for a number of new Windows Vista APIs, including `FileDialog`, `IFileOpenDialog` and `IFileSaveDialog`.

Data model

At the heart of Family.Show is its object model which is found in the **FamilyShowLib** project. The different views in Family.Show (i.e. Diagram, Family Data View, Photo/Story View, etc.) all use the same underlying data.

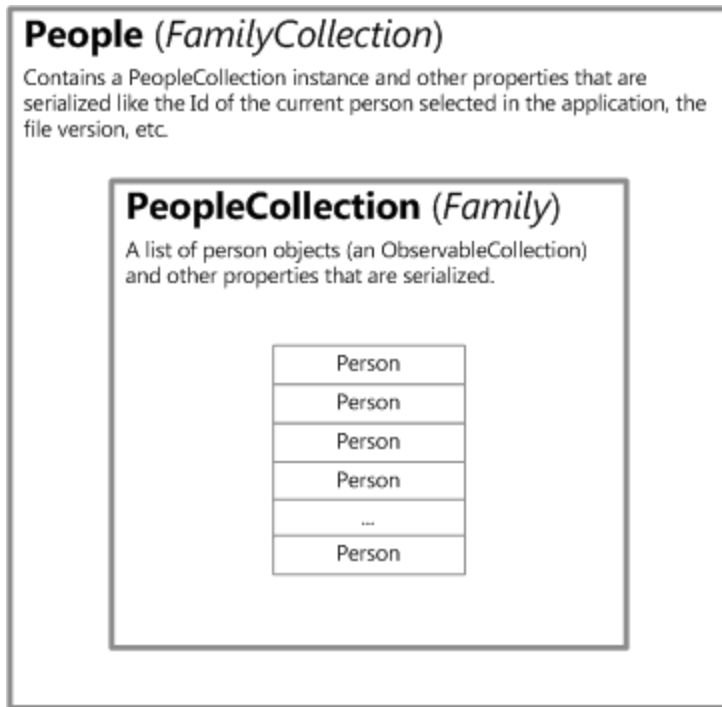


Figure 14 - People Data Model

The basic building block is the Person class. A list of people in a family is contained in a class called PeopleCollection. The PeopleCollection is then contained in another class called People. This is due to constraints with XML Serialization which is explained later in this document.

Shared Data

Instances of the People and PeopleCollections are shared by making them public static members of the **App** class (App.xaml.cs).

```
// The main list of family members that is shared for the entire application.  
// The FamilyCollection and Family fields are accessed from the same thread,  
// so suppressing the CA2211 code analysis warning.  
public static People FamilyCollection = new People();  
public static PeopleCollection Family = FamilyCollection.PeopleCollection;
```

Listing 22 - Shared Data

Views that use these collections to display data can then refer to the collection as **App.Family**, usually specifying it as a **DataContext** for one or more controls.

The class diagrams, included in the Family.Show solution, show all of the classes that make up the data model.

Person and People

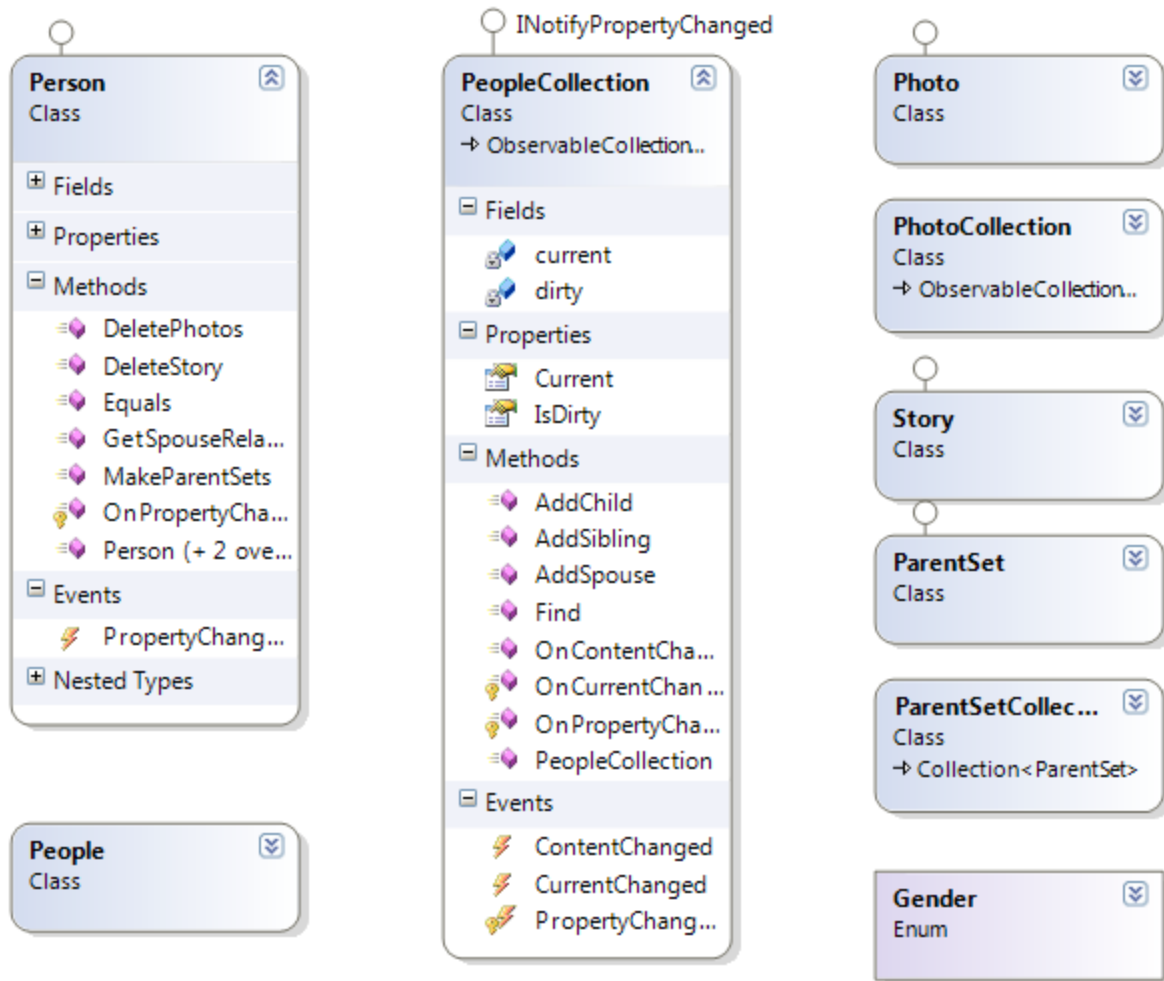


Figure 15 - Class Diagram, Person and People

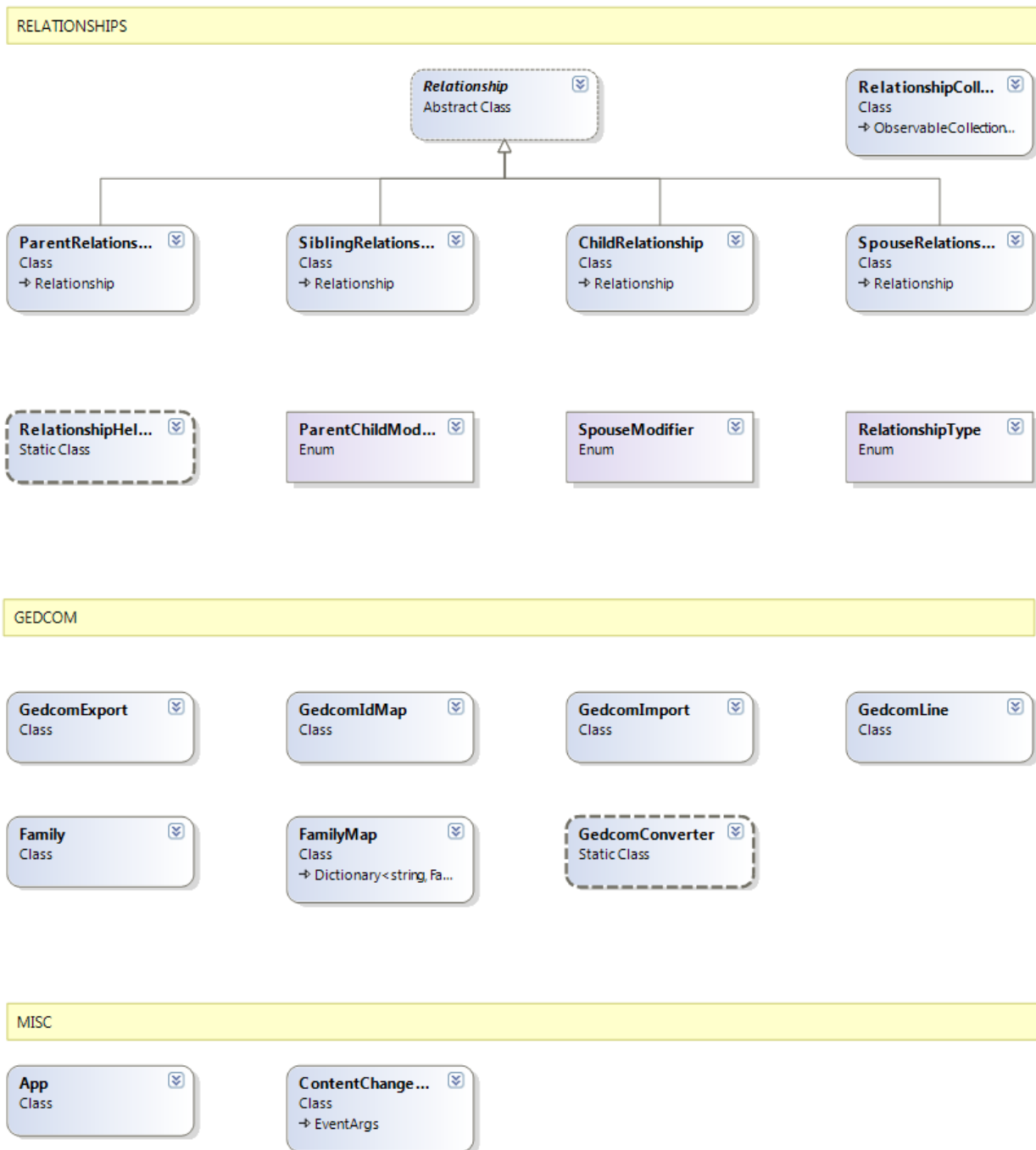


Figure 16 - Class Diagram, cont.

As shown in the class diagram, there are four types of classes in this project: the People, Relationships, GEDCOM, and Miscellaneous classes.

Person Class

Each person is represented as a **Person** object that contains properties that describe itself like first name, last name, gender, and birth date. Some of these properties are calculated from other properties

and are read-only properties. For example, the Name property is calculated from the FirstName and LastName properties.

```
/// <summary>
/// Gets the person's name in the format FirstName LastName.
/// </summary>
public string Name
{
    get
    {
        string name = "";
        if (!string.IsNullOrEmpty(firstName))
            name += firstName;
        if (!string.IsNullOrEmpty(lastName))
            name += " " + lastName;
        return name;
    }
}
```

Listing 23 - Calculated Property

INotifyPropertyChanged interface

The **Person** class implements **INotifyPropertyChanged**. This means that when a property within the class is changed, binding targets are notified of that change so they can be updated. Property change notification is a powerful concept that is used throughout the application. You can see it in action by modifying the First Name textbox in the Details control and observing the Header TextBlock change as you type.

```
public class Person : INotifyPropertyChanged, IEquatable<Person>
...
    /// <summary>
    /// Gets or sets the name that occurs first in a given name
    /// </summary>
    public string FirstName
    {
        get { return firstName; }
        set
        {
            if (firstName != value)
            {
                firstName = value;
                OnPropertyChanged("FirstName");
                OnPropertyChanged("Name");
                OnPropertyChanged("FullName");
            }
        }
    }
}
```

```

...
#region INotifyPropertyChanged Members

/// <summary>
/// INotifyPropertyChanged requires a property called PropertyChanged.
/// </summary>
public event PropertyChangedEventHandler PropertyChanged;

/// <summary>
/// Fires the event for the property when it changes.
/// </summary>
protected virtual void OnPropertyChanged(string propertyName)
{
    if (PropertyChanged != null)
        PropertyChanged(this, new PropertyChangedEventArgs(propertyName));
}

#endregion

```

Listing 24 - INotifyPropertyChanged

In Family.Show a lot of two-way binding is used so that a change to the source or target updates the other. This simplifies editing since, when you see the person's details by clicking the 'Details' button, you can edit the data without having to first go into an edit mode.

PeopleCollection class

A collection of Person objects is referred to as a **PeopleCollection**. The **PeopleCollection** inherits from **ObservableCollection** to provide notifications when new Person objects are added or removed from the collection.

```
public class PeopleCollection : ObservableCollection<Person>, INotifyPropertyChanged
```

Listing 25 - PeopleCollection

The PeopleCollection exposes methods to add new Person objects to itself based on their relationships.

People class and XML Serialization

The **People** class contains the PeopleCollection Instance. It also contains the method for loading and saving the PeopleCollection through XML serialization. It also keeps track of the currently selected person.

The people class exists because of the way XML Serialization works. We wanted to serialize the PeopleCollection class directly along with a few important properties inside of the PeopleCollection class. However, properties contained in a class that derives from a collection class are not serialized.

Circular references were also a problem that we had to overcome when serializing to XML. As mentioned earlier, a person has a collection of relationships, which itself is a collection of people. A

person in the relationship collection also has a reference back to the original person. The XML Serializer would fail in this case.

The solution was to be discerning about *what* was serialized. For relationships, instead of serializing a series of Person objects, only the person's ID was serialized. When the data was deserialized, the list relationship list is reconstructed using the IDs.

Photos and Stories

Photo and Story are classes to represent the files associated with a Person object. The Person object contains the properties for PhotoCollection and Story. The photos and stories have built-in methods that know how to save to disk.

The photos and story for a person are displayed on the PersonInfo user control. The Story is displayed using a FlowDocumentReader. The FlowDocumentReader provides built-in functionality for text layout, paging, scrolling, and zooming. Stories are edited with the RichTextBox control. The photos are displayed with the Image control.

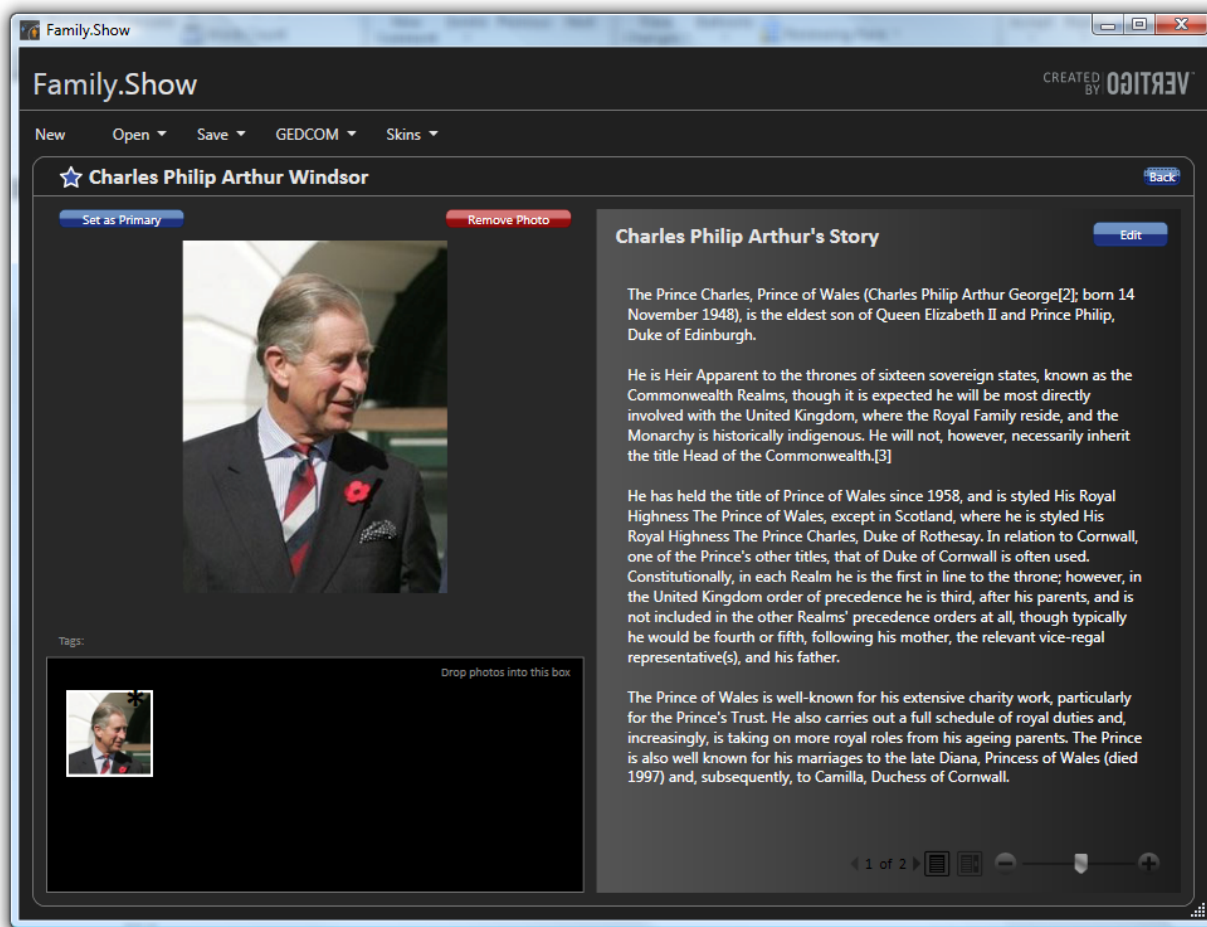


Figure 17 - Photo's and Stories View (PersonInfo)

Relationships

Each Person object contains a RelationshipCollection. The RelationshipCollection is a collection of Relationship object, which is an abstract class for describing the kinship between person objects. Relationship objects know their type of relationship and who the relation is to. There are four Relationship types: Parent, Child, Spouse, and Sibling.

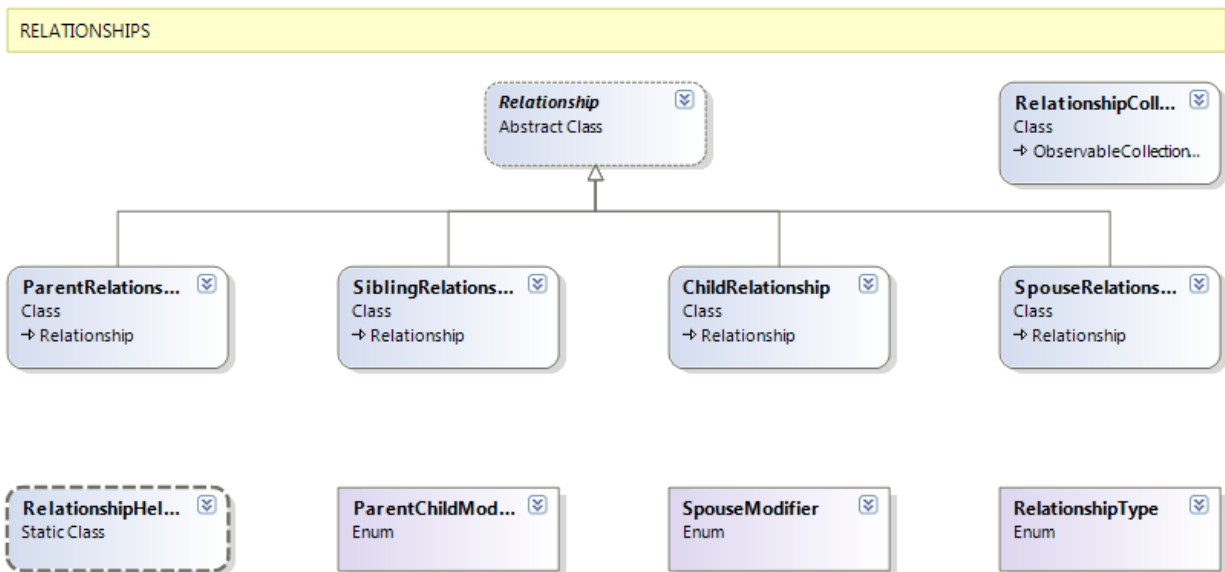


Figure 18 - Relationship Object Model

For example, Prince Charles has five relationships or connections.

Charles Windsor

- **SpouseRelationship** to Diana Spencer
- **ParentRelationship** to Philip Mountbatten
- **ParentRelationship** to Elizabeth Windsor
- **ChildRelationship** to William Windsor
- **ChildRelationship** to Henry Windsor

To make accessing subtypes of the relationship collection easier, we created calculated properties for Children, Parents, Spouses, Siblings, and HalfSiblings.

```
/// <summary>
/// Accessor for the person's children
/// </summary>
[XmlIgnore]
```

```

public Collection<Person> Children
{
    get
    {
        Collection<Person> children = new Collection<Person>();
        foreach (Relationship rel in relationships)
        {
            if (rel.RelationshipType == RelationshipType.Child)
                children.Add(rel.RelationTo);
        }
        return children;
    }
}

```

Listing 26 - Relationship Properties

RelationshipHelper class

To coordinate adding people and their relationships, we created the RelationshipHelper class. This class contains the relationship logic and rules for adding people and how they relate to each other.

For example, when adding a child, we need to add the child to the person's spouse and make the children have sibling relationships to each other as demonstrated in the code below.

```

/// <summary>
/// Performs the business logic for adding the Child relationship between the person
/// and the child.
/// </summary>
public static void AddChild(PeopleCollection family, Person person, Person child)
{
    // Add the new child as a sibling to any existing children
    foreach (Person existingSibling in person.Children)
    {
        family.AddSibling(existingSibling, child);
    }

    switch (person.Spouses.Count)
    {
        // Single parent, add the child to the person
        case 0:
            family.AddChild(person, child, ParentChildModifier.Natural);
            break;

        // Has existing spouse, add the child to the person's spouse as well.
        case 1:
            family.AddChild(person, child, ParentChildModifier.Natural);
            family.AddChild(person.Spouses[0], child, ParentChildModifier.Natural);
            break;
    }
}

```



```
}
```

Listing 27 - AddChild Method

GEDCOM classes

GEDCOM is a standard file format in the genealogical community that is used to exchange data between genealogy applications. You can read more about GEDCOM at <http://en.wikipedia.org/wiki/GEDCOM>. Family.Show supports importing and exporting data to the GEDCOM 5.5 format.

The class **GedcomImport** imports data by converting the GEDCOM file to a temporary XML file and uses XPATH to parse the data. The class **GedcomExport** exports data by iterating through the people collection and writing out individual and family sections.

ClickOnce

The application, available at www.vertigo.com/familyslow, was deployed using Microsoft's ClickOnce technology. Since publishing capabilities are built into Visual Studio, it was easy to deploy. For users running Vista, the installation is really one (or two) clicks. For those running Windows XP, users must have .NET 2.0 and .NET 3.0 which are not included with the operating system. While ClickOnce has built-in support for helping users to install pre-requisites like this, the experience can be a bit lengthy the first time they try to install Family.Show. Also, we had trouble getting ClickOnce to help deploy .NET 3.0 for Windows XP users. The .NET 3.0 extensions for Visual Studio are a CTP build, so we imagine this will be fixed in the final release.

One thing to be aware of when publishing your application using ClickOnce is the difference between the **Publishing Location** and the **Installation URL**. The Publishing Location is used to copy the files to the right location. At Vertigo, we used a UNC path to a folder on our network as the Publishing Location. This internal folder was also available via HTTP to the public. Because we initially didn't specify the correct, external Installation URL when we published the application, Vertigo employees could install the application but those outside our network could not. Public installations were looking for machines on our network. It makes a lot of sense now, but at the time it was easy to overlook.

Extending Family.Show

We tried to make the Family.Show sample as useful as possible and the team discussed a lot of ideas that did not make it into the sample. The code is available and can be modified, so feel free to add your own features and make this even more useful to the development community and genealogy community.

Here are some features we discussed that we would have implemented if we had more time:

- Improve performance when working with larger data sets. Some genealogy files are quite large with thousands of individuals.

- Use a virtual list when panning around the diagram, nodes are created and destroyed as they move onto and off the display surface.
- Explore previewing of “hidden nodes” in the diagram so that the user doesn’t need to shift their context if they want to see a spouse’s family for example.
- From a visual design perspective, explore size/color/shapes/layout in terms of making relationships in the diagram more “readable.”
- Add the ability to control the data displayed in the diagram (e.g. show/hide dates, toggle between icons and photos, etc.).
- Process files on a separate thread so the user interface remains responsive and displays the progress to the user. This includes opening and saving Family.Show files and importing and exporting GEDCOM files.
- Create another visualization that displays everyone in one giant tree.
- Allow modifying relationships by dragging the connector lines.
- Support adoptions and other family scenarios.
- Support different versions of GEDCOM, including the new 6.0 XML format.
- Support more properties in GEDCOM. Currently, some properties such as notes are ignored.
- Allow storing data on a website instead of the local file system to allow people to share and edit the same data.
- Support merging of data from other Family.Show files as well as GEDCOM data.
- Provide full printing support and provide different reports that can be printed. This includes allowing the user to print the complete family tree.
- Save and restore more settings such as the window state and position.

More information

We found the following books, websites and blogs useful when we were learning WPF and writing the Family.Show application.

Books

- Windows Presentation Foundation Unleashed by Adam Nathan
- Applications = Code + Markup by Charles Petzold
- Essential Windows Presentation Foundation by Chris Anderson

Links

- Tim Sneath’s blog (<http://blogs.msdn.com/tims>)
- The Microsoft Expression team’s [blog](#)
- [Expression Blend tutorial](#) at Lynda.com
- [Expression Design](#) tutorial
- [Performance Profiling Tools](#) for WPF
- Karsten Januszewski’s blog (<http://blogs.msdn.com/karstenj/default.aspx>)
- Mike Swanson [WPF Tools and Controls](#) post

- Kevin' Moore's blog (<http://work.j832.com/>) and his [Bag-O-Tricks](#)
- Channel 9's [Learning WPF](#)