Gradebook Sample App

***Summary**: The goal of this project is to demonstrate how to share code across project types by using a Portable Class Library between a traditional Windows* Desktop application and a Windows Store application. We chose the scenario of a gradebook for tracking students, assignments, and grades, and then developed the application accordingly.*

The Challenge: How to share code across clients and platforms?

When building an application, the developer is faced with a myriad of choices around the various technologies and tools to use. Among these, the developer must determine the format of the data objects that make up the core of the application, how to apply rules and business logic consistently, when and where to refactor and reuse logic, and the techniques for achieving promised efficiencies.

The gradebook sample application showcases how to use Portable Class Libraries to demonstrate sharing logic across application platform targets including Windows Desktop and the Windows Store. The app then follows the Model-View-ViewModel (MVVM) pattern to bind to the data. We use this approach to create classes and objects that can be reused in various client applications.

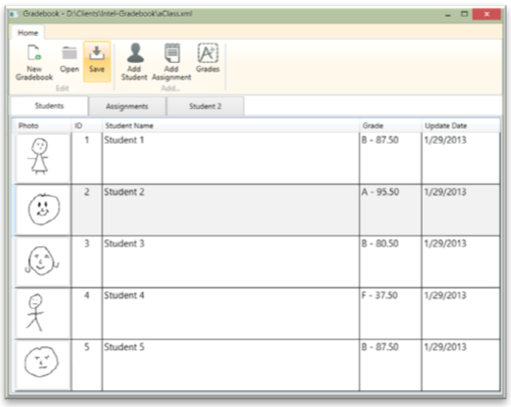
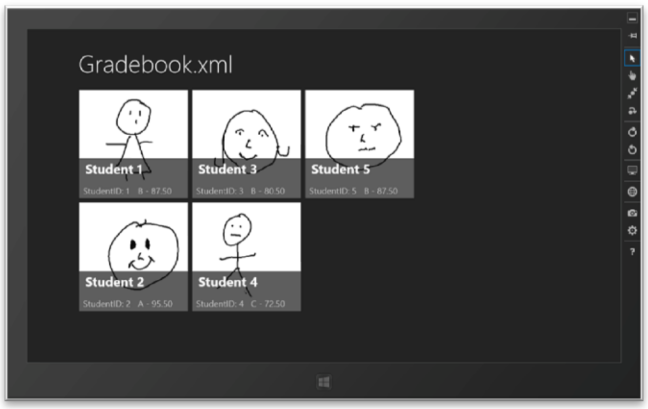


Photo	ID	Student Name	Grade	Update Date
	1	Student 1	B - 87.50	1/29/2013
	2	Student 2	A - 95.50	1/29/2013
	3	Student 3	B - 80.50	1/29/2013
	4	Student 4	F - 37.50	1/29/2013
	5	Student 5	B - 87.50	1/29/2013

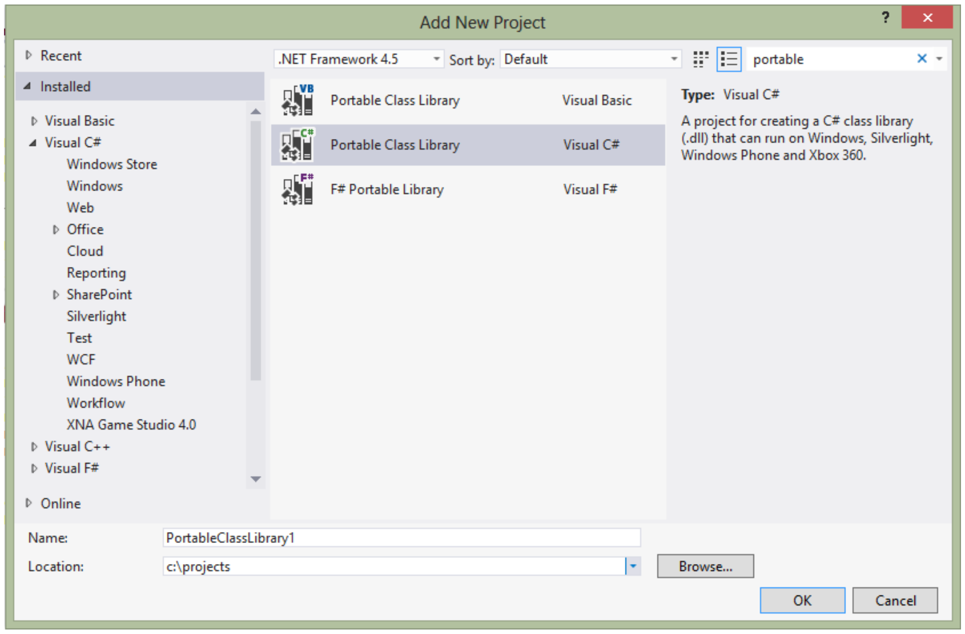


The application tracks students, assignments, and grades associated with a student. For the purposes of this example, we want to show how Portable Class Libraries allow us to leverage code written once across platforms and clients.

Working with Portable Class Libraries

Portable Class Libraries (PCL) are project types we can work with to encapsulate logic that we want to be able to share across projects and platforms. The new project type is included in Visual Studio* 2012, but is available as a package that can be added to a previous version. MSN offers an overview of working with PCL (http://msdn.microsoft.com/en-us/library/gg597391(v=VS.110).aspx), but the key thing to know is that you can create a library that can target various platforms including Windows Store, Windows Phone 7.0 and later, Silverlight* 4 and 5, and the Windows Desktop with .NET Framework 4 or later.

To add a Portable Class Library to your solution in Visual Studio, right click the solution file, add a new project to your solution and choose the type “Portable Class Library.” The output will be an assembly that you can include in your client projects. Note that because it is a separate assembly, you have the option to create the library in a different language than what your client is written in. You can create the library in C#, Visual Basic*, or F#.



.NET Framework 4.5 Sort by: Default portable

Portable Class Library	Visual Basic
Portable Class Library	Visual C#
F# Portable Library	Visual F#

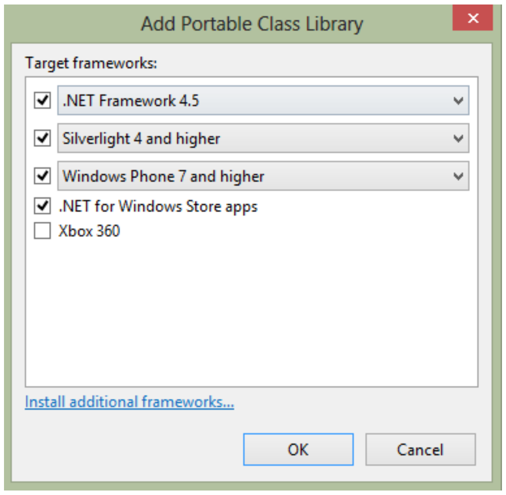
Type: Visual C#

A project for creating a C# class library (.dll) that can run on Windows, Silverlight, Windows Phone and Xbox 360.

Name: PortableClassLibrary1

Location: c:\projects

Once the PCL has been added, you will be presented with some choices for the target platform you wish to support. The resulting library will include the subset of assemblies that are common to all the targeted platforms and allow you to write code that can be shared across all.



Target frameworks:

- [✓] .NET Framework 4.5
- [✓] Silverlight 4 and higher
- [✓] Windows Phone 7 and higher
- [✓] .NET for Windows Store apps
- [ ] Xbox 360

The Model

Because we wanted to build an application that can stand alone and not be dependent on external services or processes, we used local files for storing the data. XML is a standard way to do this. We could have chosen other formats like JSON, CSV, or some native representation of the data, but working with XML has the advantage of clarifying the schema and structure of the data, as well as being a well-established format.

LINQ to XML

The key is to decide how the data will be serialized and to choose a technology that makes it easier. Language Integrated Query, or LINQ, supports creating expressions that allow us to manipulate collections of data in memory. It also has some great facilities for working with XML documents and fragments. The **XDocument** class allows us to parse an input stream and also provides functionality to save the resulting structure to disk. For example, to create an **XDocument** object from a stream, use the **Load** method, which takes as input a **Stream** object, a **TextReader**, or an **XMLReader**, and creates an XML structure as output.

```
var xDoc = XDocument.Load(inStream);
```

We could also have chosen to use **XDocument.Parse**, which will perform the same operation but with a String input, such as what you might get as a return from a call to a web service. Because we are working with file streams, we chose the former. For an overview of working with LINQ to XML, see http://msdn.microsoft.com/en-us/library/bb387061.aspx.

The classes we are using are sometimes referred to as Plain Old Compiler Objects (POCO), which, as simple base classes, use the core data types to represent the objects. For example, the class we are using for assignments is this:

```
public class Assignment
{
    public int AssignmentId { get; set; }
    public string Title { get; set; }
    public string Description { get; set; }
    public int PossibleScore { get; set; }
    public DateTime AssignedDt { get; set; }
    public DateTime DueDt { get; set; }
}
```

For our gradebook example we created classes for each of the objects including students, assignments, and grades, and then also a support class for gradebook, which contains collections of the other 3 classes.

The ViewModel

One of the strengths of the MVVM pattern is that it enables us to develop a ViewModel, or a “model of the view,” that abstracts the data from the presentation and includes the ability to communicate between the data storage and the presentation with events, commands, and data. This is accomplished by implementing an interface called **INotifyPropertyChanged** that will send a trigger to the presentation layer when the data layer is changed and vice versa. In working with an XAML-based client that takes advantage of this notification, it is common to use an **ObservableCollection** that implements this, as well as supports collection operations like adding, deleting, and modifying items within the collection.

```
    public class GradebookDataSource : INotifyPropertyChanged
    {

        public ObservableCollection<Student> Students { get; set; }
        public ObservableCollection<Assignment> Assignments { get; set; }
        public Gradebook Gradebook { get; set; }
    ...

         public event PropertyChangedEventHandler PropertyChanged;

         public void NotifyPropertyChanged(string propertyName)
         {
            if (PropertyChanged != null)
               PropertyChanged(this,
                                 New PropertyChangedEventArgs(propertyName));
         }
     }

    }
```

Next we added methods to the ViewModel to support serialization of our data. For parsing the data, we can iterate thru the elements, attributes, and nodes within the XDocument object and instantiate our classes. For example, we can read the list of assignments stored in the file by looking at the structure of the XML and recognizing that the values we are interested in lie in the Attributes of the XML.

```
var myAssignments = from n in xDoc.Descendants("AssignmentsItem")
    select new Assignment
    {
        AssignmentId = Int32.Parse(n.Attribute("AssignmentId").Value),
        AssignedDt = DateTime.Parse(n.Attribute("AssignedDt").Value),
        Title = n.Attribute("Title").Value.ToString(),
        Description = n.Attribute("Description").Value.ToString(),
        PossibleScore = Int32.Parse(n.Attribute("PossibleScore").Value),
        DueDt = DateTime.Parse(n.Attribute("DueDt").Value)
    };
```

To save our data structures to an XML format, we use a similar technique, but in this case we create an XDocument object where the constructor includes the option to create additional objects for **XElements**, **XNodes**, and **XAttributes**. The code to go from our objects saved in the local collection **Students** into an **XDocument** format is listed below.

```
var xDoc = new XDocument(
    new XElement("GradebookItem"
        , from s in Students
            select new XElement("StudentsItem",
              new XAttribute("StudentId", s.StudentId.ToString()),
              new XAttribute("StudentName", s.StudentName.ToString()),
              new XAttribute("ImagePath", s.ImagePath),
              new XAttribute("UpdateDt", s.UpdateDt.ToString())
                , from g in s.Grades
                  select new XElement("GradeItem",
                    new XAttribute("GradeId", g.GradeId.ToString()),
                    new XAttribute("StudentId", g.StudentId.ToString()),
                    new XAttribute("AssignmentId", g.AssignmentId.ToString()),
                    new XAttribute("GradeTitle", g.GradeTitle),
                    new XAttribute("Score", g.Score.ToString()),
                    new XAttribute("GradeDt", g.GradeDt.ToString())
                 )
            )
            , from a in Assignments
                select new XElement("AssignmentsItem",
                  new XAttribute("AssignmentId", a.AssignmentId.ToString()),
                  new XAttribute("Title", a.Title),
                  new XAttribute("Description", a.Description),
                  new XAttribute("PossibleScore", a.PossibleScore.ToString()),
                  new XAttribute("AssignedDt", a.AssignedDt.ToString()),
                  new XAttribute("DueDt", a.DueDt.ToString())
                )
            )
    );
```

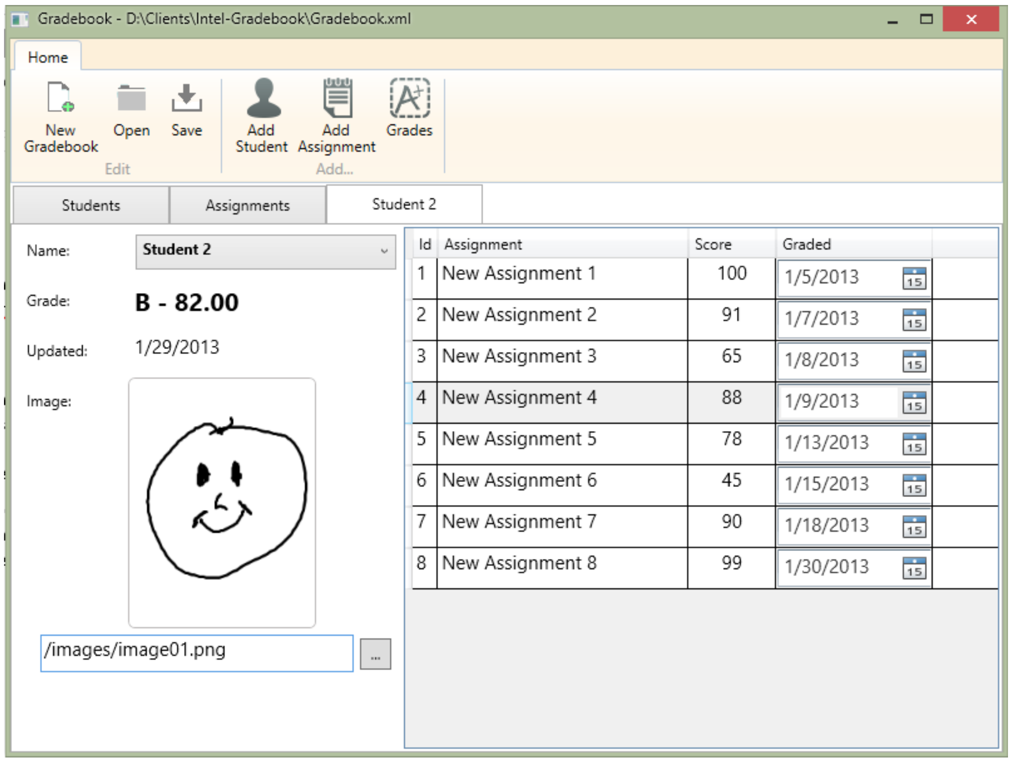
The resulting library includes classes that define our data structures and a ViewModel class **GradebookDataSource** that implements **INotifyPropertyChanged** and will work well with XAML-based clients.

Creating the Desktop Client

The first client we added is built using the Windows Presentation Foundation (WPF) model, which uses the Extensible Markup Language (XAML) to represent the user interface, and binding logic from the MVVM pattern to connect various data elements to the controls. The strength of XAML is that it is very expressive in giving the developer and designer complete control over the presentation and layout of the interface.

We built the gradebook desktop application to work with **DataGrid** controls on various tabs that are bound to the data collections in our model. Tabs are included for viewing the lists of students and assignments, and the detail of a particular student's grades.

We chose to include a ribbon with buttons for adding new students, assignments, and grades, as well as the standard file operations of creating a new gradebook, opening an existing one, and saving the current gradebook.



Name: Student 2

Grade: **B - 82.00**

Updated: 1/29/2013

Image:

/images/image01.png

Id	Assignment	Score	Graded
1	New Assignment 1	100	1/5/2013
2	New Assignment 2	91	1/7/2013
3	New Assignment 3	65	1/8/2013
4	New Assignment 4	88	1/9/2013
5	New Assignment 5	78	1/13/2013
6	New Assignment 6	45	1/15/2013
7	New Assignment 7	90	1/18/2013
8	New Assignment 8	99	1/30/2013

When the application is run, a new **GradebookDataSource** object is instantiated, and the **DataContext** of the tabs and the **ItemSource** of the **DataGrids** are set. Also, because we have not yet selected a student, the initial **DataContext** of the student detail grid is set to null.

```
private void ResetDatasource()
{
    dc = null;

    dc = new GradebookDataSource();

    dc.Students = new
        System.Collections.ObjectModel.ObservableCollection<Student>();
    dc.Assignments = new
        System.Collections.ObjectModel.ObservableCollection<Assignment>();

    gridStudents.ItemsSource = dc.Students;
    gridAssignments.ItemsSource = dc.Assignments;

    gridStudentDetail.DataContext = null;
}
```

Setting the **DataContext** and the **ItemsSource** on container controls allows us to use binding of the child controls to the data source. For example, the Student Name is bound to a TextBlock on the details tab with the following syntax:

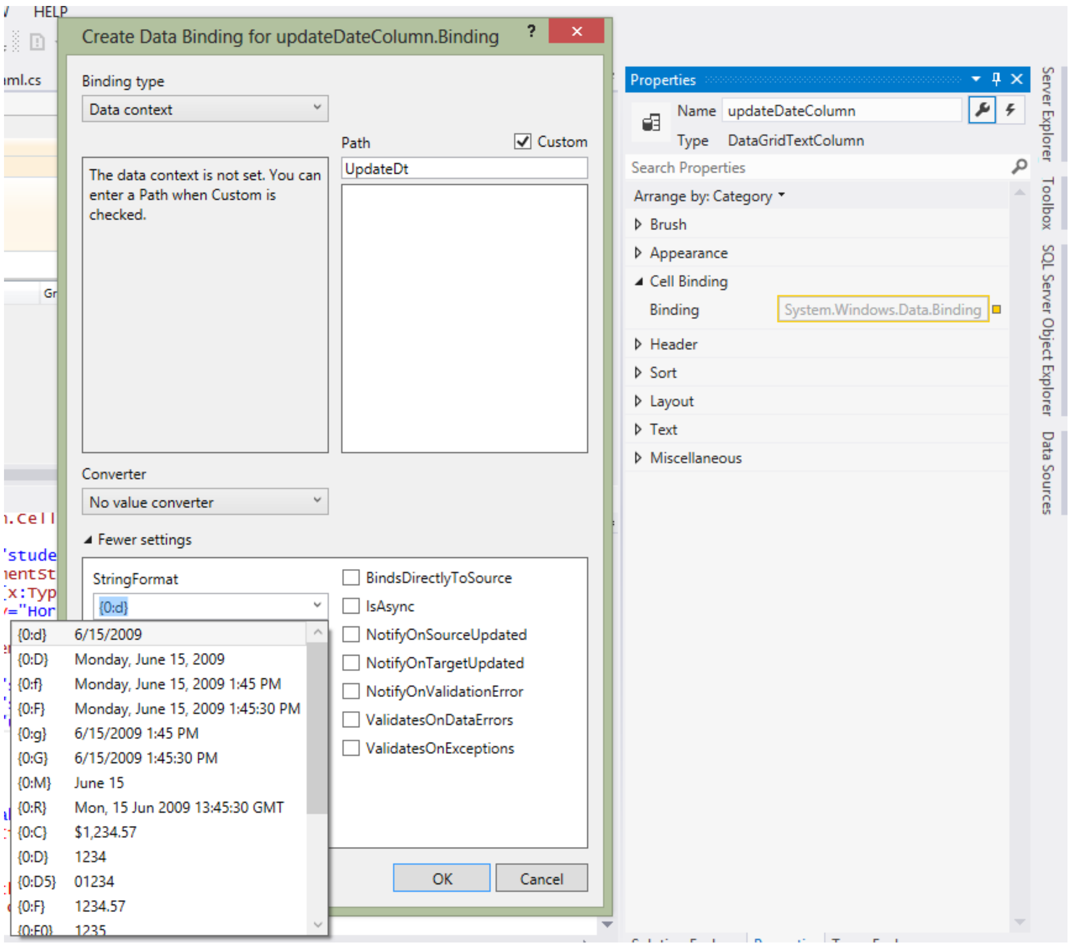
```
<TextBlock Text="{Binding StudentName}" FontWeight="Bold" />
```

In WPF we have a rich set of options for data binding including formatting the data according to its type, such as date, default values, and more. In the Student **DataGrid** we used the following binding syntax to specify the

Student Name, Grade and Updated Date columns. Notice that the Grade column has a default value, specified as a FallbackValue. Similarly, we are able to specify binding for the Updated Date column which has **IsReadOnly** set to “True”, and includes a **StringFormat** to bind the data to a Date format.

```
    <DataGridTextColumn x:Name="studentNameColumn"
         Binding="{Binding StudentName}" Header="Student Name" Width="*"/>
    <DataGridTextColumn x:Name="studentGradeColumn"
         IsReadOnly="True" Header="Grade" Width="100"
         Binding="{Binding GPA, FallbackValue='U', Mode=TwoWay}"/>
    <DataGridTextColumn x:Name="updateDateColumn" IsReadOnly="True"
         Header="Update Date" Width="125"
         Binding="{Binding UpdateDt, StringFormat=\{0:d\}}" />
```

One of the nice things about working in Visual Studio’s XAML editor is that the binding dialog for WPF applications, which you can access from the properties window, includes a dialog that has a dropdown of common string formats!



Create Data Binding for updateDateColumn.Binding

Binding type: Data context

The data context is not set. You can enter a Path when Custom is checked.

Path: UpdateDt [✓ Custom]

Converter: No value converter

Fewer settings

StringFormat: {0:d}

{0:d}	6/15/2009
{0:D}	Monday, June 15, 2009
{0:f}	Monday, June 15, 2009 1:45 PM
{0:F}	Monday, June 15, 2009 1:45:30 PM
{0:g}	6/15/2009 1:45 PM
{0:G}	6/15/2009 1:45:30 PM
{0:M}	June 15
{0:R}	Mon, 15 Jun 2009 13:45:30 GMT
{0:C}	$1,234.57
{0:D}	1234
{0:D5}	01234
{0:F}	1234.57
{0:F0}	1235

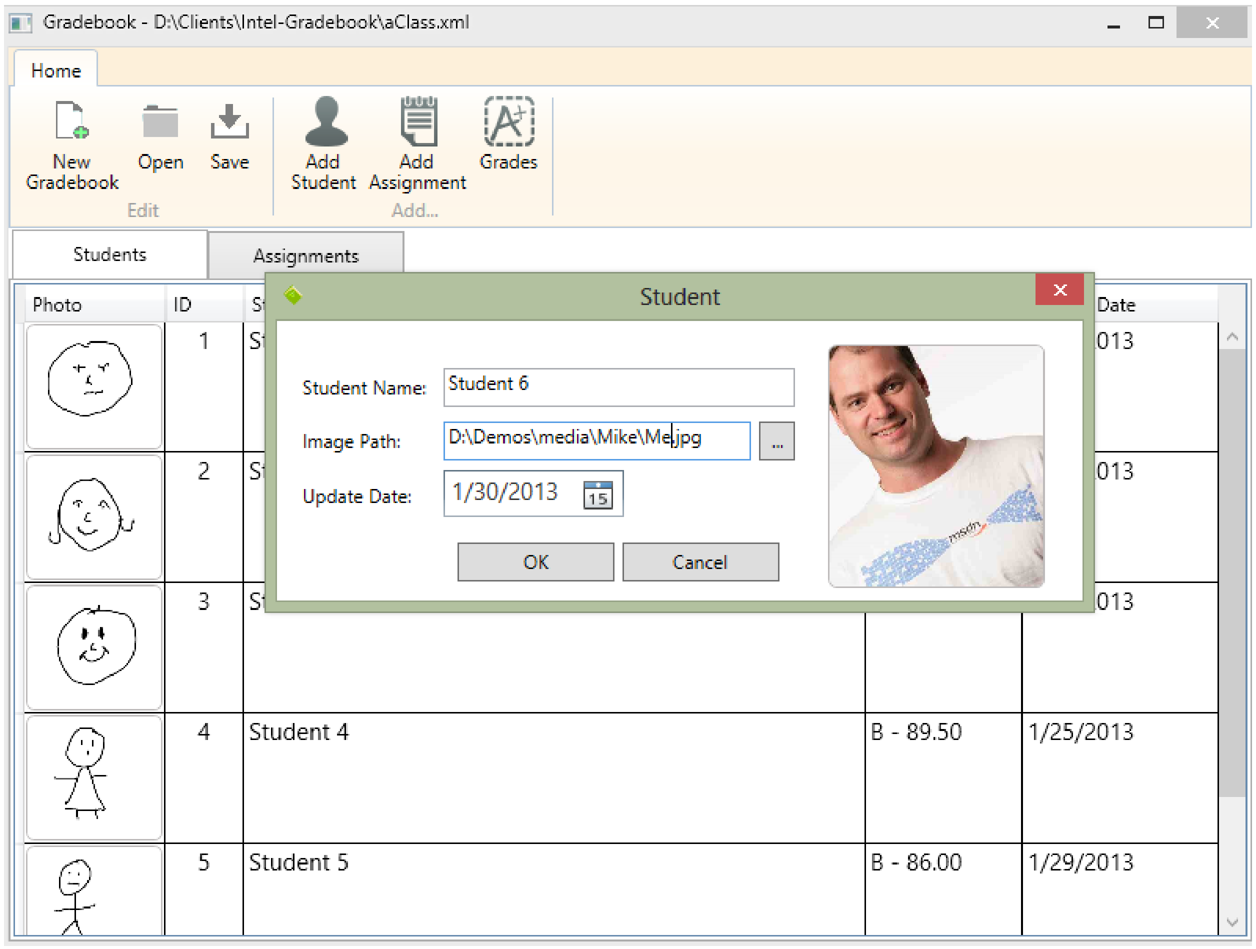
☐ BindsDirectlyToSource
☐ IsAsync
☐ NotifyOnSourceUpdated
☐ NotifyOnTargetUpdated
☐ NotifyOnValidationError
☐ ValidatesOnDataErrors
☐ ValidatesOnExceptions

Properties

Name: updateDateColumn
Type: DataGridTextColumn

Search Properties
Arrange by: Category
- Brush
- Appearance
- Cell Binding
  - Binding: System.Windows.Data.Binding
- Header
- Sort
- Layout
- Text
- Miscellaneous

The rest of the code in the application includes dialogs to add new students and assignments and manage grades. We also included support for customizing the student pictures with Drag and Drop. On the Student Detail tab and the Add Student dialog, the property **AllowDrop** is set to “True,” and the events for **DragEnter** and **Drop** have handlers added, which then update the **ImagePath** property of the Student object.



Gradebook - D:\Clients\Intel-Gradebook\aClass.xml

Student Name: Student 6

Image Path: D:\Demos\media\Mike\Me.jpg

Update Date: 1/30/2013

Photo	ID	St		Date
	1	St		013
	2	St		013
	3	St		013
	4	Student 4	B - 89.50	1/25/2013
	5	Student 5	B - 86.00	1/29/2013

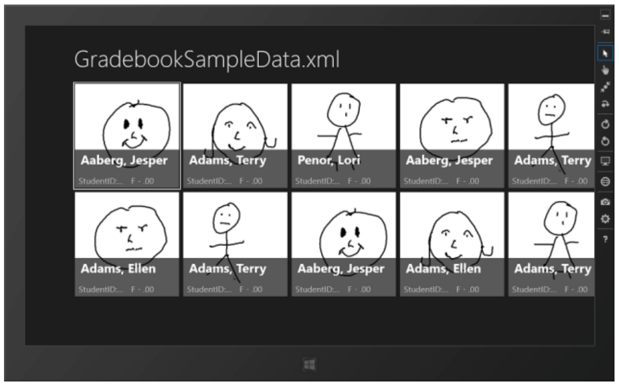
The code for this is below.

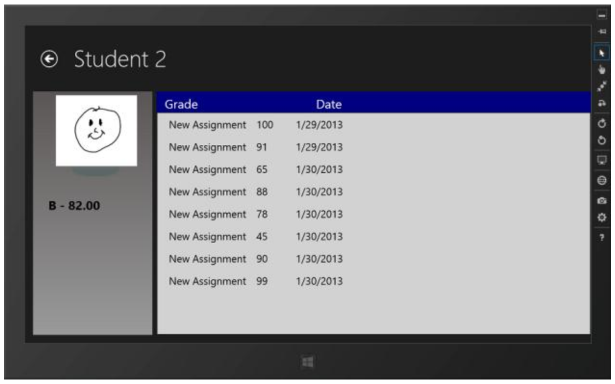
```
private void Image_DragEnter_1(object sender, DragEventArgs e)
{
    if (e.Data.GetDataPresent(DataFormats.FileDrop))
    {
        e.Effects = DragDropEffects.Copy;
    }
    else
        e.Effects = DragDropEffects.None;

}

private void Image_Drop_1(object sender, DragEventArgs e)
{
    if (e.Data.GetDataPresent(DataFormats.FileDrop))
    {
        //  If the file is a .bmp file, set it as the background image
        foreach (string filename in
                 (string[])e.Data.GetData(DataFormats.FileDrop))
        {
            var ext = System.IO.Path.GetExtension(filename);
            if (ext == ".jpg" || ext == ".jpeg" || ext == ".png")
            {
                SetImage(filename);
            }
        }
    }

}
```

Finally we added logic to update the student's GPA when grades are entered. When the user makes a change to the Grade **DataGrid**, a **RowEditEnding** event is handled to update the data source and the new grade is reflected in the bound controls.

Building the Windows Store Application

Our Windows 8 Store app will use the PCL we created to work with the same files as the WPF desktop application.





Grade		Date
New Assignment	100	1/29/2013
New Assignment	91	1/29/2013
New Assignment	65	1/30/2013
New Assignment	88	1/30/2013
New Assignment	78	1/30/2013
New Assignment	45	1/30/2013
New Assignment	90	1/30/2013
New Assignment	99	1/30/2013

Windows Store apps are built on the new Windows Runtime framework that allows developers to choose between XAML with C# or VB.Net languages, HTML5 + JavaScript*, or C++ with XAML. The resulting application runs in a sandbox-type environment with a security model where the application specifies the capabilities up front, and limits the application to that part of the machine.

Visual Studio 2012 also provides a number of templates and features that make building Windows Store applications easier, but it helps to understand their features and how they work. Some of the key things the templates manage for us include:

- Data binding via the MVVM pattern to our data source using a BindableBase class that implements **INotifyPropertyChanged**.
- Support for various presentation modes with the **LayoutAwarePage** class.
- **SuspensionManager** makes it easier to work with app data and Page State data and is wired into the application lifecycle events.
- Converters for handling visibility and more.

The Windows Store project types include a Blank app, which only includes the minimal items; the Grid App ,which includes the core template classes and a Group, Group List and Detail pages; and the Split App, which includes two-level hierarchy pages for Group List and Detail.

Even if you select the empty application template, you still have the option to use the various page template types including Split Page, Items Page, Item Detail Page, Grouped Items Page, Group Detail Page, and Basic Page. When these are added, you are prompted whether you want to add the core files and classes of the templates.

For our gradebook application we used an Items Page and a Basic Page. The Items page includes a grid with tiles that can be templated to allow the user to select a student. The basic page includes the wiring for the MVVM and the flexibility to choose our detailed layout. For the Items page, the DataTemplate had to be modified to reflect our Data Source.

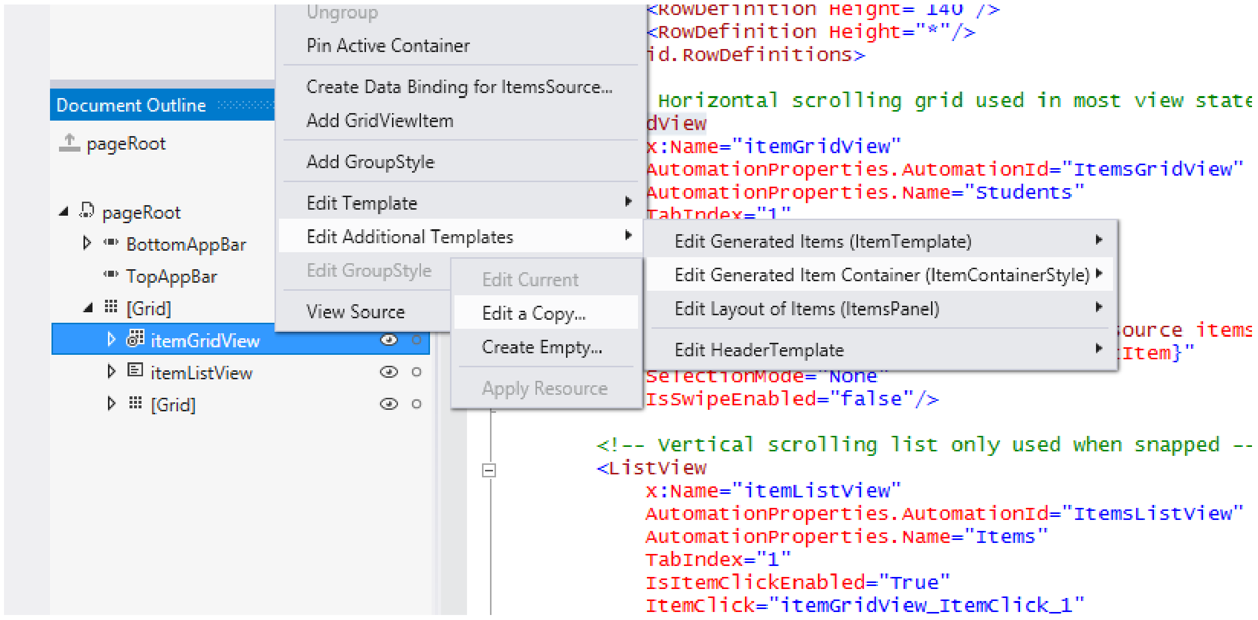
The first step is wiring up the View Model to the application. This is done by adding a reference to the PCL and adding logic to set the **DefaultViewModel** to our data in the **LoadState** event handler. Because we are working with XML files, we are using the native FilePicker to open a file and setting our references accordingly.

```
Gradebook myGradebook;

private async void OpenFile()
{
    var myPicker = new FileOpenPicker();

    myPicker.SuggestedStartLocation = PickerLocationId.ComputerFolder;
    myPicker.FileTypeFilter.Add(".xml");

    var myFile = await myPicker.PickSingleFileAsync();
    if (myFile != null)
    {
        GradebookDataSource dc = new GradebookDataSource();

        System.IO.Stream myStream = await myFile.OpenStreamForReadAsync();

        myGradebook = dc.OpenGradebook(myStream);

        this.DefaultViewModel["Students"] = myGradebook.Students;
        this.DefaultViewModel["Assignments"] = myGradebook.Assignments;
        this.DefaultViewModel["Gradebook"] = myGradebook;

        myGradebook.Filename = myFile.Name;
        this.pageTitle.Text = myGradebook.Filename;
    }
}
```

Next we customize the templates for displaying the data to work with our data structures.

```
<DataTemplate x:Key="myStudentItem">
    <Grid HorizontalAlignment="Left" Width="250" Height="250">
        <Border Background="{StaticResource
                ListViewItemPlaceholderBackgroundThemeBrush}">
            <Grid>
                <Image Source="/assets/student_32.png"
                       Height="200" Width="200"
                       HorizontalAlignment="Center"
                       VerticalAlignment="Center"/>
                <Image Source="{Binding ImagePath}"
                       Stretch="UniformToFill"
                      AutomationProperties.Name="{Binding ImagePath}"/>
            </Grid>
        </Border>
        <StackPanel VerticalAlignment="Bottom"
                    Background="{StaticResource
                        ListViewItemOverlayBackgroundThemeBrush}">
            <TextBlock Text="{Binding StudentName}"
                    Foreground="{StaticResource
                        ListViewItemOverlayForegroundThemeBrush}"
                    Style="{StaticResource TitleTextStyle}"
                    FontSize="30" TextWrapping="Wrap" FontWeight="Bold"
                    Height="37" Margin="15,15,15,0"/>
            <StackPanel Margin="10" Orientation="Horizontal">
                <TextBlock Text="{Binding StudentId,
                     ConverterParameter='StudentID: \{0\} ',
                     Converter={StaticResource StringFormatConverter}}"
                     Foreground="{StaticResource
                  ListViewItemOverlaySecondaryForegroundThemeBrush}"
                     Style="{StaticResource CaptionTextStyle}"
                     FontSize="20" TextWrapping="NoWrap"
                     HorizontalAlignment="Left" Width="126"/>
                <TextBlock Width="126" HorizontalAlignment="Right"
                     Text="{Binding GPA}" Foreground="{StaticResource
                  ListViewItemOverlaySecondaryForegroundThemeBrush}"
                     Style="{StaticResource CaptionTextStyle}"
                     FontSize="20" TextWrapping="NoWrap"/>
            </StackPanel>
        </StackPanel>
    </Grid>
</DataTemplate>
```

As you can see, the **DataTemplate** works closely with defined styles for determining how the elements are displayed. These styles are initially included in the **StandardStyles.xaml** file found in the Common folder of the templates. One way for getting to the right template from the Visual Studio XAML designer is to use the Document Outline pane, right click on the appropriate object, and edit the templates. If you choose to edit a copy, the XAML is copied to the Resources section of the current page.



```
<RowDefinition Height="140"/>
<RowDefinition Height="*"/>
id.RowDefinitions>

 Horizontal scrolling grid used in most view state
dView
x:Name="itemGridView"
AutomationProperties.AutomationId="ItemsGridView"
AutomationProperties.Name="Students"
TabIndex="1"
                                                    ource items
                                                    Item}"
SelectionMode="None"
IsSwipeEnabled="false"/>

    <!-- Vertical scrolling list only used when snapped --
    <ListView
        x:Name="itemListView"
        AutomationProperties.AutomationId="ItemsListView"
        AutomationProperties.Name="Items"
        TabIndex="1"
        IsItemClickEnabled="True"
        ItemClick="itemGridView_ItemClick_1"
```

The key changes are the binding names for the display properties. I also included a default background image in case the image referenced in the data is not in an accessible location to the Windows Store app.

Differences from desktop apps

One of the limits we encountered is access to the local file system. The **Windows.Storage** features allow us to work with **KnownFolders**, including the local installation directory and the various libraries, but require access tokens to other files on the system, which we can get by working with the **FilePicker** dialogs and calling the **MostRecentlyUsedList** to return the token. See http://msdn.microsoft.com/en-us/library/windows/apps/jj655411.aspx for more details.

Another thing you'll notice is that the XAML binding subset doesn't support **StringFormat** in the binding syntax out of the box, but we can implement our own by creating a Converter and adding a class that implements the **IValueConverter** interface.

```
public class StringFormatConverter : IValueConverter
{
    public object Convert(object value, Type targetType,
                          object parameter, string language)
    {
        // No format provided.
        if (parameter == null)
        {
            return value;
        }

        return String.Format((String)parameter, value);
    }

    public object ConvertBack(object value, Type targetType,
                              object parameter, string language)
    {
        return value;
    }
}
```

Future enhancements
Looking forward, it would be nice to make the Windows Store application fully editable, allowing the user to add students, assignments, and work with the complete set of data. To make this work, we would need to add pages for editing a student and their grades. The **DataGrid** control available in WPF is not yet available in Windows Store, so we would need to write our own to support editing the grades in a ListView.

Summary
Building reusable code is possible by using Portable Class Libraries. They allow developers to share code between projects targeting diverse platforms.