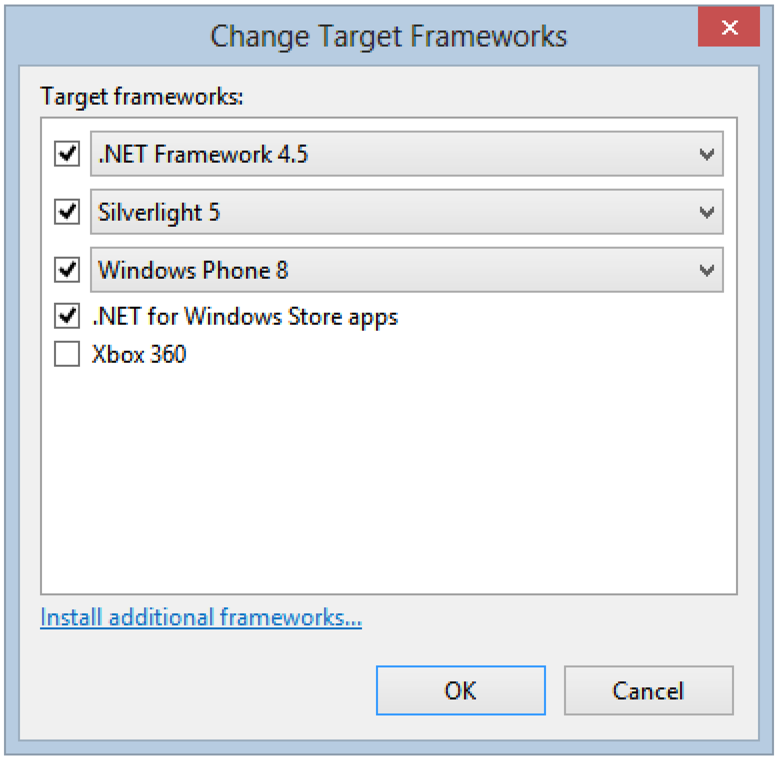
Strategies for Sharing Code between Windows Phone 8 and Windows 8 Applications

by Benjamin Day
http://www.benday.com

Visual Studio 2012 is out and we can write apps for Windows Store and now (finally) we have access to the Windows Phone 8 SDK and Windows Phone 8 devices. We've been hearing and reading for a while that the platforms are very similar to code for and after doing some coding for both platforms I only kind of agree. Windows Store applications for Windows 8 are written using C#, XAML, and WinRT. Windows Phone 8 is still basically Silverlight.

The real big question on everyone's mind is just how much code you can actually share between your Windows Store apps and your Windows Phone apps. The answer is that you can actually share a lot of code but – get ready to not be surprised – like most things in software, you have to plan for it. That planning will almost definitely require you to abstract away a lot of details and use interface-driven programming.

There are 3 main ways to share code: 1) portable class libraries, 2) multiple projects using linked code, and 3) branching & merging via your source control repository. A Portable Class Library is a project type in Visual Studio 2012 that lets you target multiple frameworks from a single project. In the case of Figure 1 -- Choose the frameworks for a Portable Class Library, you can see that the project targets version 4.5 of the .NET Framework, Silverlight 5, Windows Phone 8, and .NET for Windows Store apps. Basically, what's going on here is that Visual Studio 2012 now becomes smart enough to know what is the lowest common denominator for all of these frameworks, will tell you what isn't valid, and will generate an assembly that can be referenced by all these platforms.

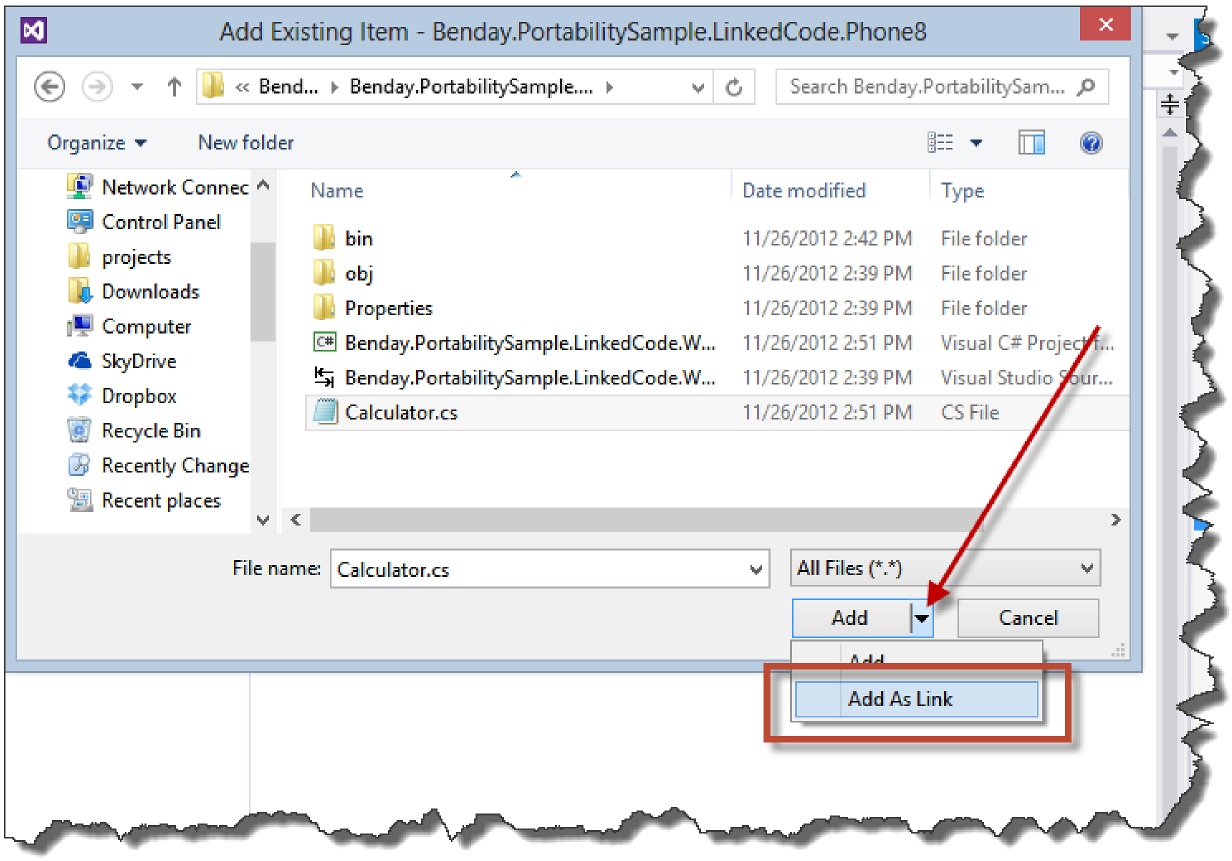

Change Target Frameworks

Target frameworks:

- [✓] .NET Framework 4.5
- [✓] Silverlight 5
- [✓] Windows Phone 8
- [✓] .NET for Windows Store apps
- [ ] Xbox 360

Figure 1 -- Choose the frameworks for a Portable Class Library

The second way of sharing is via linked code. The idea here is that you have two different project files that point to the same classes on disk. Let's say you have a Windows Phone 8 assembly project and a Windows Store assembly project – respectively, Project A and Project B. You'll first create a class in Project A and then you'll go to Project B, choose Add, and then Add An Existing Item. When the Add Existing Item dialog pops up, you select the class file, and then instead of clicking the Add button, you'll choose Add As Link instead. The Add As Link option isn't obvious and it's hidden in a little dropdown menu off of the Add button as shown in Figure 2 -- Add an existing item using Add As Link. Once you've created the link, any changes you make to that file in one project is instantly reflected in that same file in the other project.



Name	Date modified	Type
bin	11/26/2012 2:42 PM	File folder
obj	11/26/2012 2:39 PM	File folder
Properties	11/26/2012 2:39 PM	File folder
Benday.PortabilitySample.LinkedCode.W...	11/26/2012 2:51 PM	Visual C# Project f...
Benday.PortabilitySample.LinkedCode.W...	11/26/2012 2:39 PM	Visual Studio Sour...
Calculator.cs	11/26/2012 2:51 PM	CS File

File name: Calculator.cs

Figure 2 -- Add an existing item using Add As Link

The third way to share code is using the branching and merging functionality of a source control repository such as Team Foundation Server 2012. This approach is going to look similar to the linked code version of code sharing. You'll have two different projects (Project A and Project B) that target different platforms and they'll have similar files in them. The main difference is that you'll have two separate copies of the shared files on your disk and any changes you make won't automatically propagate between projects – you'll need to perform a Merge in source control in order to move files between projects. The blessing here is that you can track exactly what is shared and you can decide when you choose to move sync code between the projects. If you use the linked code approach, changes you make in Project A might work on that platform but break the app in Project B. The curse with the branching and merging approach is that it's a little cumbersome.

For any of these approaches, there are upsides and downsides. Sharing via branching and merging appeals to my inner control freak but it's cumbersome and is almost definitely going to put you into the dreaded and labor intensive "cherry pick merge" model. The Linked Code approach has a fair number of downsides, too. First off, it offends my inner Application Lifecycle Management (ALM) Control Freak because changes instantly appear in both projects. Secondly, it's exactly the same code in both projects – right down to the namespace. So, you'll have two classes in your solution that have exactly the same namespace and class name. That nags at me because the default namespace for Project A and Project B might not be the same. In Project A it might be TheAwesomeApp.WinPhone.MyClass and in Project B you might want it to TheAwesomeApp.WinStore.MyClass but you'll be stuck with the original namespace and now the naming convention in Project B won't match all the other classes in that project. I know that it's a little OCD of me but clean, well-organized code is easier to maintain.

Sharing using the Portable Class Library approach for sharing feels like it's the cleanest and most polished way to go. Like the rest of the sharing approaches, you'll need to write platform agnostic, lowest common denominator, non-partisan code but you don't have any branch/merge headaches and you're dealing with literally the same binary for each platform rather than having to think about a similar but not-quite-the-same version. It's the same binary. Done and done.

Finding the Lowest Common Denominator (LCD)

Three quick thoughts on Portable Class Libraries (PCLs) and the Lowest Common Denominator (LCD) problem. Thought #1: if you're writing a PCL, you have to write code that works on all the platforms targeted by that PCL. That set of code that works on all the platforms is the LCD. Thought #2: platform-specific, non-Portable exe's and assemblies can reference a PCL but a PCL can only reference another PCL. Thought #3: The big things that are different between the platforms are networking code, data-access code, and which APIs are required to be asynchronous. Those three things govern the shape of your application architecture because of what can reference what and what you need to be abstract.

When I write a XAML-based application with C#, I use a layered architecture that is comprised of the following layers:

1) The XAML user interface layer: this is the platform-specific user interface that targets the Phone or the Windows Store or Silverlight or .NET/WPF. This project generates an EXE or a XAP.
2) The Model-View-ViewModel (MVVM) presentation layer: This project contains my ViewModels. The ViewModels are the code-based abstraction of everything that is displayed on a particular screen. My XAML-based user interfaces are going to have references to instances of the ViewModels and use data binding to move information to and from these classes and the running user interface that the user will manipulate.
3) The Model layer: This project contains the Model classes (aka. the "business objects") that contain my business logic. These Models are what get saved to and from my database or file system storage or web services.
4) The Data Access layer: This project could also be called the Persistence Layer because it is primarily concerned with persisting (aka. saving) my application data to whatever my persistent store is. A persistent store could be a database or a service or the file system or local storage. Basically, anyplace that I can store data so that that data doesn't disappear when I pull the plug on my app. The classes in the project commonly use the Repository Pattern and the Adapter Pattern to save and load instances of the Models to/from my persistent store.

5) The Common code layer: This project will contain any common code that can be referenced by any of the other layers. This project contains the C# interfaces that are implemented by my Models, ViewModels, Adapters, and Repositories.

In the simplest form, the user interface references the ViewModels which reference the Models which references the Data Access layer and everything references Common (see Figure 3 -- Layers and their references). When you're sharing code across platforms, the ViewModels, Models, and Common code are the same and the XAML UIs and the Data Access layers are completely different. This means that the ViewModel, Model, and Common projects are all going to be PCLs and the other projects will be platform-specific.

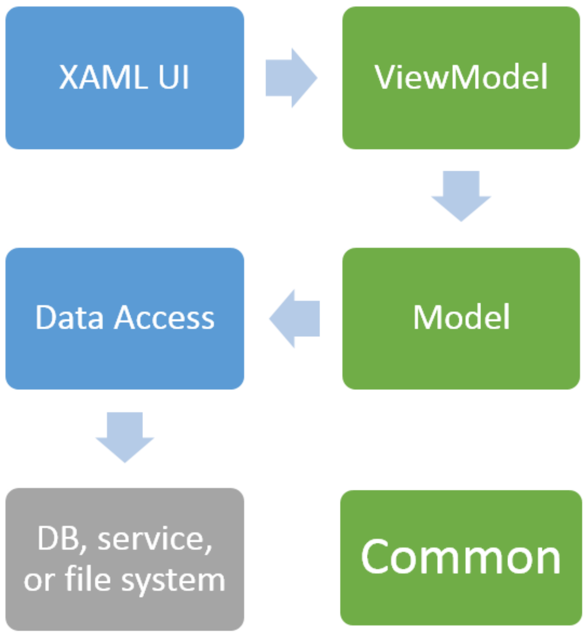


Figure 3 -- Layers and their references

Remember that thought about what can reference a PCL and what a PCL is allowed to reference? Anything can reference a PCL but a PCL can only reference another PCL. For the XAML UI to the ViewModel the reference will be fine because that's platform-specific to a PCL. What becomes more difficult is that reference from the Model to Data Access because that's going from a PCL to a platform-specific DLL and that's not allowed. (Game over.)

There's another nifty little detail about PCLs and interfaces and this is why you're going to care about interface-driven programming if you want to share code between platforms. Let's say that you define an interface called IPersonDataAccess in a PCL. This interface represents all the operations that your system needs in order to save and retrieve Person model classes. Better yet, let's say that IPersonDataAccess represents all the operations required to save and retrieve instances of IPerson models. You can now create platform-specific classes that implement IPersonDataAccess. Here's the cool part – let's say that your PersonViewModel class depends on IPersonDataAccess to save and retrieve data. As long as PersonViewModel references IPersonDataAccess rather than a platform-specific implementation, you can pass in platform-specific instances of IPersonDataAccess to

PersonViewModel and it will be perfectly happy as shown in Figure 5 -- ViewModelLocator that creates an instance of PersonViewModel and passes in a platform-specific instance of IPersonDataAccess.

```
/// <summary>
/// Interface for person data access
/// defined in a portable class library
/// </summary>
public interface IPersonDataAccess
{
    Task<IList<IPerson>> GetAll();
    Task<IPerson> GetById(int id);
    Task Save(IPerson saveThis);
}

/// <summary>
/// Interface for person model
/// defined in a portable class library
/// </summary>
public interface IPerson
{
    int Id { get; set; }
    string FirstName { get; set; }
    string LastName { get; set; }
    string EmailAddress { get; set; }
}
```

Figure 4 -- IPersonDataAccess and IPerson interfaces in a PCL

```
/// <summary>
/// A ViewModelLocator helps keep track of the instances of a
/// ViewModel in an application.  They can be either PCLs or
/// platform-specific.  This code represents a platform-specific
/// ViewModelLocator.
/// </summary>
public class ViewModelLocator
{
    public ViewModelLocator()
    {
        // create an instance of PersonViewModel
        // and pass in an instance of
        // WindowsPhonePersonDataAccess
        m_ViewModelInstance = new PersonViewModel(
            new WindowsPhonePersonDataAccess());
    }

    private PersonViewModel m_ViewModelInstance;
    public PersonViewModel ViewModelInstance
    {
        get { return m_ViewModelInstance; }
    }
}
```

Figure 5 -- ViewModelLocator that creates an instance of PersonViewModel and passes in a platform-specific instance of IPersonDataAccess

```
/// <summary>
/// This view model class is defined in a PCL and depends on
/// IPersonDataAccess to save and load data.
/// </summary>
public class PersonViewModel : INotifyPropertyChanged
{
    /// <summary>
    /// Constructor that requires an instance of
    /// the IPersonDataAccess dependency to be "injected"
    /// </summary>
    /// <param name="dataAccess"></param>
    public PersonViewModel(IPersonDataAccess dataAccess)
    {
        m_DataAccess = dataAccess;
    }

    private IPersonDataAccess m_DataAccess;

    public async Task LoadById(int id)
    {
        // call into the instance of IPersonDataAccess
        // to get some data
        var result = await m_DataAccess.GetById(id);

        // do something with the result
    }

    // ...

    public event PropertyChangedEventHandler PropertyChanged;
}
```

Figure 6 -- the PersonViewModel class depends on IPersonDataAccess and takes an instance of IPersonDataAccess on the constructor

```
/// <summary>
/// Implementation of IPersonDataAccess that is
/// specific to the Windows Phone platform.
/// </summary>
public class WindowsPhonePersonDataAccess : IPersonDataAccess
{
    public Task<IList<IPerson>> GetAll()
    {
        IList<IPerson> theResults = new List<IPerson>();

        // do some work that is specific to Windows Phone

        return new Task<IList<IPerson>>(() => theResults);
    }

    public Task<IPerson> GetById(int id)
    {
        IPerson theResult = null;

        // do some work that is specific to Windows Phone

        return new Task<IPerson>(() => theResult);
    }

    public Task Save(IPerson saveThis)
    {
        // do some work that is specific to Windows Phone

        return new Task(null);
    }
}
```

Figure 7 -- a Windows Phone-specific implementation of IPersonDataAccess

This pattern is known as Dependency Injection. If Class A relies on another Class B or Interface B in order to do work, that is known as a dependency. Rather than having Class A internally create an instance of Class B, you create that instance of Class B outside of Class A and then "inject" it into Class A's constructor. If Class A depends on Interface B, then this means that Class A is interface-driven. Being interface-driven and using Dependency Injection helps us to bridge the divide between the PCLs and the platform-specific projects and re-use code. Remember, our problem in Figure 3 -- Layers and their references was referencing out platform-specific Data Access code from the Models and ViewModels that live in PCLs. If you code against interfaces you can just inject the versions of the classes that are specific to Windows Phone or Windows Store apps.

Factoring in Asynchronous Code into your Lowest Common Denominator

Over the last handful of years, asynchronous operations have become more and more common. While we've known for a while that we can improve the user experience and performance of our applications by making certain operations asynchronous, this generally wasn't required until Silverlight and Windows Phone 7 (WP7). In Silverlight and WP7, any networking calls had to be async so if your application needed to read or write data over a network, you needed to adjust your application architecture to work with async. That doesn't sound like that would cause too much difficulty but, since async network calls returned instantly, that makes using methods with ordinary return values pretty much impossible. Windows Phone 8 (WP8) still has the same requirement that network calls be asynchronous but this now gets a lot easier with async, await, and System.Threading.Tasks. Those sharp and jagged edges of asynchronous coding aren't nearly as scary anymore.

Enter Windows RT and Windows Store applications for Windows 8. If you're a XAML and C# programmer, WinRT is going to look very familiar but there is a lot of new stuff. New classes. New methods. New project types. And lots more asynchronous code. Once again, just like in Silverlight and WP8, async, await, and System.Threading.Tasks makes working with this asynchronous code fairly painless but if your goal is to share code between platforms, this is where you need to start paying attention. Assuming that you're doing interface-driven programming, you'll need to craft your interfaces so that they can adequately wrap (aka "hide") the details of the most asynchronous platform that you want to target from your PCLs. If you're targeting both Windows Phone 8 and Windows Store application for Windows 8, the Windows Store apps are definitely going to require more asynchronous code. As a result, if you have the option to do so, you'll probably want to write the Windows Store version of your libraries first so that you discover as much of the required async logic as possible.

Summary

The good news is that you can definitely share code between the platforms. The bad news is that that sharing won't happen for free and you'll need to design your application in order to enable it. Using Portable Class Libraries, interface-driven programming, and Dependency Injection will help you to achieve that sharing. You might be thinking that having to build your application like this is going to be painful or difficult but I think you’ll find that there’s a silver lining for this particular cloud – you’ll be forced to follow some great best practices for application design and you’ll be pleasantly surprised at how it help make your application more maintainable.