Co-development for Windows Phone 7/8 and Windows 8 guide

This article explains how to write code that can target multiple platforms including Windows Phone 7/8, Windows 8 and beyond

Introduction

Building applications that target multiple platforms is always hard. We would like to minimize maintenance, testing and coding time in order to ship more quickly.

Each platform has its own set of APIs, it can have different capabilities, screen resolutions and all those differences creep into the code. The key is to isolate the code which is platform independent, i.e. it is the same for all platforms, and separates it from the platform specific code. This separation will minimize the written piece of code. Traditional solutions for reusing code across projects were either "copy paste pattern" which didn't help reducing the overall code size or sharing source files in multiple projects. The latter way introduced its own quirks and, as we will see later on, this technique is pretty much still used, although significantly less.

But how do we do that? One way is to structure your application using the MVVM pattern and keep views in the platform specific projects and put view models and models inside a shared library. Since ordinary class library cannot be shared between multiple platforms, there is a new project type called Portable Class Library (later referred as PCL). It can be consumed in Windows Phone 7, Windows Phone 8 and Windows 8 projects (later referred as WP7, WP8 and W8 respectively).

However, PCL are C# assemblies and as for WinRT, there is no such thing for C++ based Runtime Components. You have to share the code between them using the old way. It will be described in more detail further down this article.

Benefits of Cross Platform Code Reuse

- Reduced Development Time : Code modification will propagate to all platforms, thus reduced development time
- Ease in code maintenance : Changes in code is now handled at one place alone and then sharing the same binaries.
- Excess code can lead to more defects : Reusable design will lead to less code and in turn less defects.
- Consistent Quality
- Improved 'Time to market'
- Reduced project cost

Platform overview

	Windows Phone 7	Windows Phone 8	Windows 8
Screen size	800x480	800x480	1366x768 (minimum full)
		1280x720	1024x768 (minimum filled)
		1280x768	320x768 (minimum snapped)
Preferred screen orientation	Portrait	Portrait	Landscape
Programming languages	C#, Visual Basic, F#	C#, Visual Basic	C#
		C++/CX	C++/CX
	HTML5 + js (†)	HTML5 + js (†)	HTML5 + js + WinJS
APIs	.NET	.NET	.NET
	XNA	XNA (††)	
		DirectX (sans Direct2D)	DirectX
		WinRT (†††)	WinRT (†††)

†) HTML5 applications on Windows Phone 7 are not the same as on Windows Phone 8. There are some technical details involved and note that Windows Phone 8 SDK comes with a project template for creating Windows Phone HTML5 App. Also note that this is quite different from Windows Store applications which use WinJS. All three platforms are different and portability is quite an issue.

††) XNA is deprecated in Windows Phone 8 which only means that you cannot target Windows Phone 8 with a XNA project, only Windows Phone 7. Note that XNA applications will run on WP8 devices. If you still want to target WP8 or W8 and use XNA as a framework, you can use MonoGame, an open source implementation of XNA.

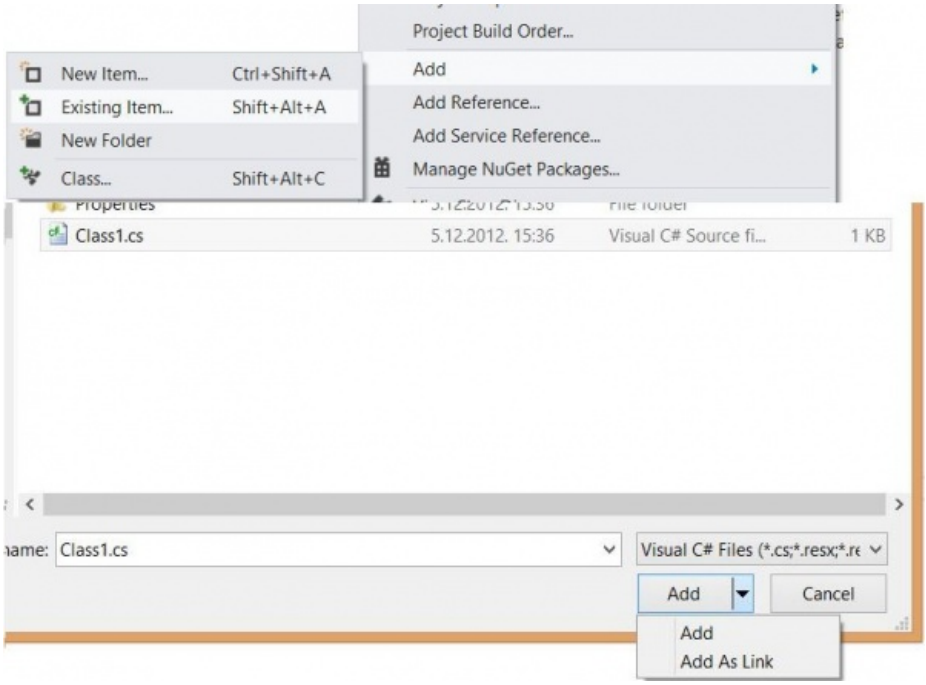
†††) Although WP8 and W8 share the same core and share a subset of Windows RT APIs, you cannot reuse components between the two platforms. More on that below.

Note: Note the multitude of different screen sizes, large differences in programming languages available. This article will focus on sharing C# and C++ code. Other languages (Visual Basic, F#, JavaScript) should have their own dedicated topics.

Sharing source files

This techniques was prevalent before PCL came along. Source code that could be shared between multiple platforms could be added to different solutions as linked file. The difference between standard file adding is that the linked file is not copied to the project, it is simply referenced as if it was an external source. You can still debug it and it works as a normal file, the only difference is that code is shared.

To add an external source file to a project without copying it to the project's folder, use the Add/Add Existing Item context menu option in Solution Explorer. Browse to the file you wish to include and instead of clicking Add, click on the small triangle next to it and select Add As Link in the dropdown menu as seen on the image below.



This way you can share assets as well. Note that linked files have small blueish symbol in the lower right corner that indicates that it isn't part of the project and that it is linked.

When you share a source file between projects that target multiple platforms, you might run into two kinds of problems:

1. The code will not compile without small changes.
2. The code will compile, but there are some runtime differences.

Let's explore those differences in more depth.

Compile time differences

Sometimes the code is *almost* the same for two or more platforms. Some functions might be renamed, they can accept different parameters or they might function differently. If you can isolate these small chunks of code, you can use preprocessor to compile the code differently depending on which project the file is included into. Let's say we want to write a download helper class and we are targeting both W8 and WP8. You can use **HttpClient** in W8 applications to download strings from the given URI, but in WP8 you must use **WebClient**. One way to implement it is with the following code:

```
using System.Threading.Tasks;

#if NETFX_CORE
using System.Net.Http;
```

```
#else
using System.Net;
#endif

namespace W8ClassLibrary
{
    public class DownloadHelper
    {
        public static async Task<string> DownloadString(string uri)
        {
#if NETFX_CORE
            var client = new HttpClient();
            return await client.GetStringAsync(uri);
#else
            var client = new WebClient();
            return await client.DownloadStringTaskAsync(new System.Uri(uri));
#endif
        }
    }
}
```

We can now write our app logic using this helper class and its methods and that code will be portable. The symbol **NETFX_CORE** is defined only in W8 projects and this is how we can enforce compile time changes to the code.

Runtime differences

Sometimes the code will compile regardless of the platform, but we need to check on which platform the code is executing in runtime. This is useful when dealing with In App Purchases or Live Tiles on Windows Phone. For example, if you are still maintaining a Windows Phone 7 application, but you wish to use the features that are provided by Windows Phone 8 OS when you detect that you are running on it, you can use reflection to access them. It is obvious that you can check which operating system version you are running on only in runtime. This is useful when you do not want to convert or work in parallel on the WP8 version of your application. Check out the following links for more information:

- Using IAP in Windows Phone 7.1 projects - code sample
- Adding Windows Phone 8 Tile functionality to Windows Phone OS 7.1 apps

You can also adapt your UI depending on the OS version and load images of different sizes or use completely different assets. Code using reflection will compile regardless of the destination platform, but once the code actually runs, you can adapt your behavior depending on the runtime differences.

To detect Windows Phone OS version, you can use the following fragment:

```
public bool IsRunningOnWP8
{
    get
    {
        return Environment.OSVersion.Version.Major >= 8;
    }
}
```

You can detect current resolution if you are compiling for Windows Phone 8 using the `Application.Current.Host.Content.ScaleFactor` property. But if the code that uses the above property is shared between WP7 and WP8 projects, it will not compile. In this case we use both compile time and runtime testing. The following function returns **ScaleFactor** value regardless of the project type it is included into.

```
public static class ResolutionChecker
{
    static int _scaleFactor = -1;
```

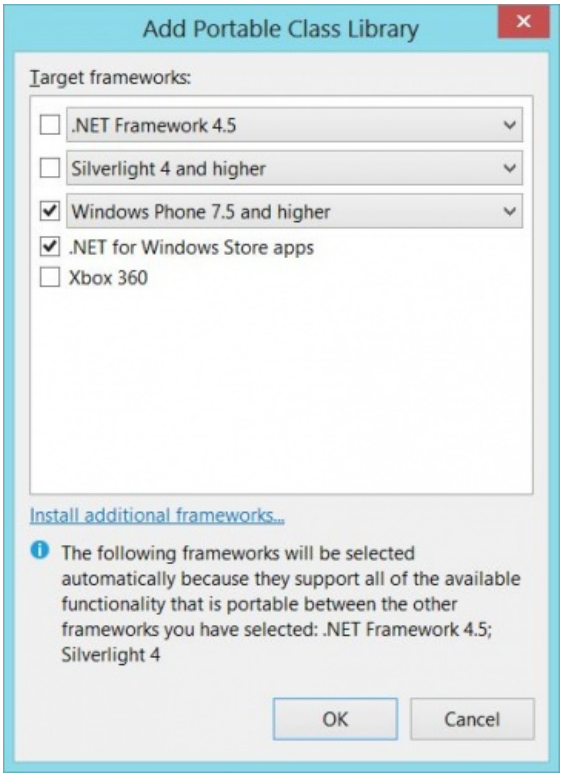
```
    public static int ScaleFactor
    {
        get
        {
            if (_scaleFactor == -1)
            {
#if WINDOWS_PHONE7
                if (Environment.OSVersion.Version.Major < 8)
                {
                    _scaleFactor = 100;
                }
                else
                {
                    var scaleFactorProperty =
Application.Current.Host.Content.GetType().GetProperty("ScaleFactor");
                    if (scaleFactorProperty != null)
                        _scaleFactor =
(int)scaleFactorProperty.GetValue(Application.Current.Host.Content, null);
                }
#elif WINDOWS_PHONE8
                _scaleFactor = Application.Current.Host.Content.ScaleFactor;
#endif

            }
            return _scaleFactor;
        }
    }
}
```

Portable Class Library

Sharing source code files will only get you so far. You still have to maintain a separate library for each new platform you wish to support. If you want to add a new class, you need to manually update each library and add the new file as a linked file. This is clearly impractical. To solve that, Visual Studio 2012 comes with a new project type called Portable Class Library.

PCL is a special type of class library that is compiled only once and can be run on multiple .NET platforms without recompiling. This means that you can put your platform agnostic code in it and reuse it across different projects that target specific platforms. You write your source once, put it in one project and you compile it to one assembly - instead of having an assembly for each platform with shared code. Since you now maintain only one version of source files, this reduces the maintenance cost usually assigned to various platform's quirks drastically and gives you flexibility when writing new apps.

When you create a PCL, you are prompted to choose which platforms you want to target. Depending on the chosen platforms, a minimal set of APIs that is present in all selected platforms is available to you. You can only add reference to other portable libraries which target the same set of APIs.



Depending on the targets you choose, you will have a limited subset of Base Class Library (from now on referred as BCL) that exists in all selected platforms and many other APIs such as XML handling, core networking, XLINQ, serialization, etc. All the infrastructure for writing MVVM code is available if you target even .NET Framework 4. This makes it easy to put your view models in portable code. It is obvious that many APIs were already present in multiple platforms and you can see the full matrix of available features on the image below.

Feature	.NET Framework	Windows Store	Silverlight	Windows Phone	Xbox 360
Core	√	√	√	√	√
LINQ	√	√	√	√	
IQueryable	√	√	√	Only 7.5	
Dynamic keyword	Only 4.5	√	√		
Managed Extensibility Framework (MEF)	√	√	√		
Network Class Library (NCL)	√	√	√	√	
Serialization	√	√	√	√	
Windows Communication Foundation (WCF)	√	√	√	√	
Model-View-View Model (MVVM)	Only 4.5	√	√	√	
Data annotations	Only 4.0.3 and 4.5	√	√		
XLINQ	Only 4.0.3 and 4.5	√	√	√	√
System.Numerics	√	√	√		

Here are the links to if you want to read more about PCL:

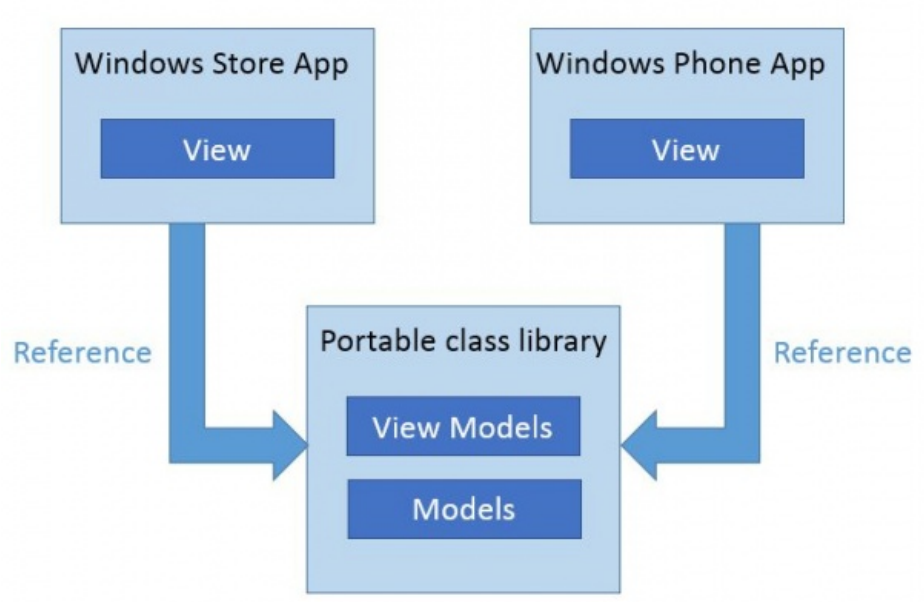
1. http://blogs.msdn.com/b/dsplaisted/archive/2012/08/27/how-to-make-portable-class-libraries-work-for-you.aspx
2. http://msdn.microsoft.com/en-us/library/gg597391.aspx
3. http://blogs.msdn.com/b/bclteam/archive/2012/05/09/announcing-portable-library-tools-2-beta-for-visual-studio-

2010.aspx

4. http://blogs.msdn.com/b/dotnet/archive/2012/07/06/targeting-multiple-platforms-with-portable-code-overview.aspx

General pattern

MVVM is the recommended pattern for writing Windows Phone and Windows RT applications. It allows for easy separation of UI code and the application logic. Once you separate them, UI code can stay in platform specific projects while the application logic can be placed into one or several PCL projects. You can use **INotifyPropertyChanged** and **ICommand** interfaces inside your portable library if you target Windows Store and Windows Phone apps. This ensures that your existing code built with MVVM pattern in mind can be made portable quickly.



Platform specific implementation

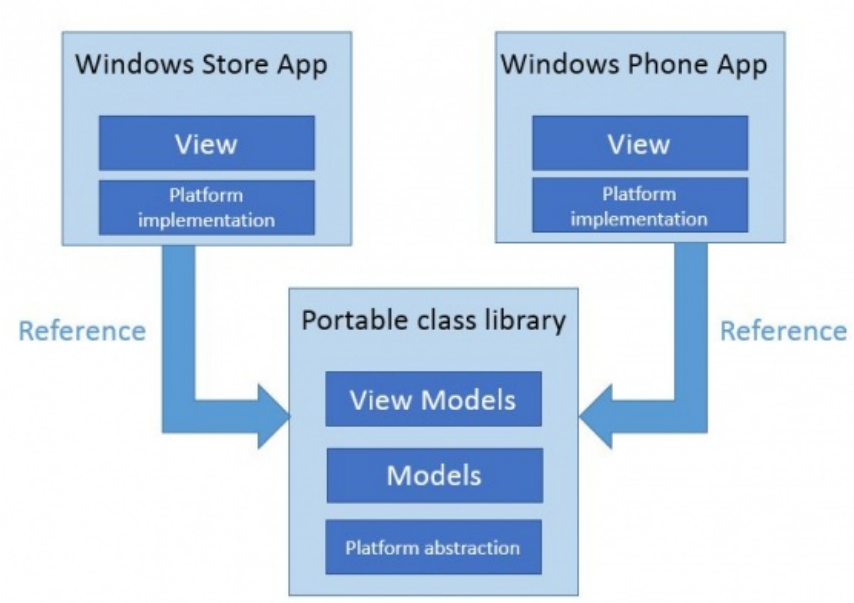
Even though you can separate application logic from the presentation layer, sometimes you need to perform an action that is present on all target platform, but the API is not portable. Since we don't want to move parts of application logic to platform specific projects, we can use abstraction pattern to solve this issue.

Consider the above mentioned **DownloadHelper** class. When we shared the code between platform specific projects, we could change which code is compiled using compiler flags. This is not available in PCL since you compile your library only once and for all platforms once. It is obvious that we can't use those APIs in the PCL, but we can abstract that functionality. Now each platform specific project will implement abstract functionality using platform specific APIs. This approach keeps PCL testable and you can still write correct application logic even though you don't call concrete APIs, you call your own instead.

Another example of functionality you might want to use, but APIs differ:

- Choosing an image on Windows 8 using **FilePicker**, but with **PhotoChooserTask** on WP
- You can get list of image folders both on WP device and Windows 8 device, but the APIs differ
- Opening URI in browser: **WebBrowserTask** on WP, **Launcher.LaunchUriAsync** on Windows 8

Note: Both **WebRequest** and **WebResponse** are portable, while **WebClient** and **HttpClient** aren't.



So how do we convert **DownloadHelper** to PCL? There are several ways to do it: IoC (Inversion of Control), Dependency Injection, using ServiceLocator. We can group solutions roughly into two categories:

- Abstracted functionality is available as singleton globally.
- Abstracted functionality is given upon construction (IoC, DI).
- Portable library dynamically determines platform and loads appropriate platform specific implementation.

Let's see how we would implement it using the second technique. Create PCL that targets both WP8 and W8 and add new class inside named **IDownloadHelper**.

```
using System.Threading.Tasks;

namespace PCL
{
    public interface IDownloadHelper
    {
        Task<string> DownloadString(string uri);
    }
}
```

Let's say that our view model is declared in the following fashion:

```
namespace PCL
{
    public class InfoViewModel
    {
        IDownloadHelper _downloadHelper;

        public InfoViewModel(IDownloadHelper downloadHelper)
        {
            _downloadHelper = downloadHelper;
        }
    }
}
```

The idea is to pass a concrete implementation upon view model construction. The concrete implementation will be supplied by platform specific project. In the WP8 project, implement the interface:

```
using System.Net;
```

```
using System.Threading.Tasks;
using PCL;

namespace WP8App
{
    public class DownloadHelper : IDownloadHelper
    {
        public async Task<string> DownloadString(string uri)
        {
            var client = new WebClient();
            return await client.DownloadStringTaskAsync(new System.Uri(uri));
        }
    }
}
```

In Windows Store app you would implement **IDownloadHelper** analogously:

```
using System.Net.Http;
using System.Threading.Tasks;
using PCL;

namespace W8App
{
    class DownloadHelper : IDownloadHelper
    {
        public async Task<string> DownloadString(string uri)
        {
            var client = new HttpClient();
            return await client.GetStringAsync(uri);
        }
    }
}
```

Async support and PCL

When you target both Windows Phone 8 and Windows Store Apps with your PCL, you can use async/await pattern and you have **Task** available. If you target Windows Phone 7, you cannot use any of them. But there is support for async/await pattern available (even though it is prerelease at the time of writing this) and you can actually write WP7 apps with async/await pattern.

How do we target all three platforms then? Unfortunately, you need to resort back to linked files. You can put portable code in PCL targeting both WP8 and W8 and create a new Class Library for WP7 with all files from PCL added as links. Unless you don't care about supporting WP7 (and I strongly recommend that you support it), this is the only way.

Note: Be aware that some classes have different names e.g. **TaskEx.Delay** (WP8) vs. **Task.Dely** (WP7).

Notable portable libraries

It is strongly recommended that you write all of your future libraries as portable libraries if possible. This makes it easier to interop later on without the need for porting it.

Some notable libraries have been converted to portable libraries:

- AutoFac
- Ninject
- Json.NET
- Reactive Extensions
- MvvmLight

Check out Portable Class Libraries Contrib for some additional features available for portable libraries.

WinRT components

This section applies only to WP8 and W8 since native code is not available on WP7 platform.

There is a great deal of API exposed as WinRT API and can be consumed in native components. This is particularly important for DirectX based applications since it is available only for native code. There is alse a great deal of other API functionality available as WinRT API like networking, proximity, In-App Purchase, sensors, location, file system, core app model and threading API.

Unfortunately, you cannot write single WinRT compoment and use it in both WP8 and W8 projects. To share code between WP8 and W8 RT components, you must use linked files. All platform specific code should be wrapped in preprocessor directives similarly to the sample shown above.

Cross Platform Code Reusability Best Practices

Writing reusable code may take more time in the analysis and design phase, but this cost is usually recovered by reducing both the total cost of development and the cost of ongoing maintenance. This section contains a few key guidelines for creating reusable code.

Keep the UI separate

.NET uses MVVM model i.e. view : view-model : model, where the view represents the UI layer.

WP8 and Windows have similar UI design guidelines for a consistent look and feel. But the way apps appear and interact with desktop and mobiles differ and hence it's good to avoid re-engineering too much on code reuse of this layer.

Reusing the application business logic

The best layer that can be part of reusable component is the business logic layer, which will remain the same irrespective of how it is presented on UI.

Reusing the data layer

This is a bit tricky, as in some cases this layer can be made reusable but in some cases it won't.

Example : Above mentioned Sales App, the database and services pulling data from it will be the same and can be a part of reusable components.

But consider an example of a photo editor application; here the data (image files) on which the business logic will act can be highly platform specific and hence in this case, this layer should not be a candidate for code reuse.

Summary

Supporting multiple platforms requires extra resources, but can be greatly reduced using Portable Class Libraries. Sharing WinRT native code is not that simple, but can be done through linked files.

Reference links

- [How to Leverage your Code across WP8 and Windows 8]

The media player is loading...

- [Create Cross-platform Apps using Portable Class Libraries]

The media player is loading...

Version Hint

Windows Phone: [[Category:Windows Phone]]
[[Category:Windows Phone 7.5]]
[[Category:Windows Phone 8]]

Nokia Asha: [[Category:Nokia Asha]]
[[Category:Nokia Asha Platform 1.0]]

Series 40: [[Category:Series 40]]
[[Category:Series 40 1st Edition]] [[Category:Series 40 2nd Edition]]

[[Category:Series 40 3rd Edition (initial release)]] [[Category:Series 40 3rd Edition FP1]] [[Category:Series 40 3rd Edition FP2]]
[[Category:Series 40 5th Edition (initial release)]] [[Category:Series 40 5th Edition FP1]]
[[Category:Series 40 6th Edition (initial release)]] [[Category:Series 40 6th Edition FP1]] [[Category:Series 40 Developer Platform 1.0]] [[Category:Series 40 Developer Platform 1.1]] [[Category:Series 40 Developer Platform 2.0]]

Symbian: [[Category:Symbian]]
[[Category:S60 1st Edition]] [[Category:S60 2nd Edition (initial release)]] [[Category:S60 2nd Edition FP1]] [[Category:S60 2nd Edition FP2]] [[Category:S60 2nd Edition FP3]]
[[Category:S60 3rd Edition (initial release)]] [[Category:S60 3rd Edition FP1]] [[Category:S60 3rd Edition FP2]]
[[Category:S60 5th Edition]]
[[Category:Symbian^3]] [[Category:Symbian Anna]] [[Category:Nokia Belle]]