CUTTING EDGE

Code reuse in WPF and Silverlight 2.

Dino Esposito – October 2008

CODE DOWNLOAD AVAILABLE AT: MSDN Code Gallery **(604 KB)**
Browse the Code Online

This column is based on a prerelease version of Silverlight 2. All information herein is subject to change.

Contents

In Silverlight 2, you use Extensible Application Markup Language (XAML) to design and render the user interface. At the same time, you leverage the built-in core CLR to process managed code within the browser. As a result, there's a strong similarity between Web-based Silverlight 2 applications and desktop Windows Presentation Foundation (WPF) applications. One goal of having a similar programming model is that it enables easy code reuse between the two. In this column, I'll discuss a few patterns to make sharing code and XAML markup between Silverlight 2 and WPF as easy as possible.

CoreCLR In Silverlight

Silverlight 2 does incorporate a CLR, but it's not the same CLR used by the other .NET applications and assemblies. The Silverlight CLR is known as CoreCLR and has been designed with different goals in mind. Designed for cross-platform interoperability, the CoreCLR can run side by side with the CLR, and it supports a different security model and a different version of the base class library. More is explained in the excellent CLR Inside Out column of August 2008 (see msdn.microsoft.com/magazine/cc721609).

Having different CLRs behind Silverlight and .NET applications means that you can't have the same assembly referenced in two projects targeting a .NET application and a Silverlight application. The main issue is with the mscorlib assembly. Silverlight needs a much smaller set of functionality for its purposes—just the core stuff. Any .NET assembly, instead, links to the standard version of mscorlib, and this is where the trouble starts.

In the sample application discussed in this column, I have an interface to share between the Windows Presentation Foundation application and the Silverlight application. The only workaround is copying the C# with the interface definition between the two projects because you can't have a commonly referenced assembly; therefore, it is desirable that in the version of the .NET Framework the functionalities in the standard mscorlib assembly are split into two parts—core stuff and desktop stuff—in order to lay the groundwork for binary compatibility between Silverlight and .NET assemblies.

WPF and Silverlight 2 Compatibility

With the introduction of Silverlight 2, XAML is poised to become the API for a new generation of UI modules. Silverlight 2 supports a subset of the full WPF framework including features for rich layout management, data binding, styles, media, animation, graphics, and templates.

However, full support for XAML in Silverlight 2 is constrained by the size of the downloadable plug-in. In just a few megabytes (less than 5MB in Beta 2), the Silverlight 2 plug-in must deliver the core CLR, a version of the Microsoft .NET Framework 3.5 that includes WPF and a subset of the Windows Communication Foundation (WCF) client platform, a XAML parser, and a number of Silverlight-specific controls.

In Silverlight 2, you won't find support for the WPF 3D graphics capabilities, and you may find that some attributes and elements have been dropped or trimmed down. At the end of the day, though, you have a compatible subset, so you can produce a Silverlight version of a moderately complex WPF user interface. Shortly, I'll return to discuss critical areas of compatibility.
Note, however, that Microsoft is also adding some new APIs in Silverlight that may not currently have a counterpart in the desktop version of WPF. The most relevant features include controls, such as the DataGrid, and classes, such as WebClient for simple GET-style network calls. These features will be added to WPF too.

Introducing Visual State Manager

Another feature that ships in Silverlight 2 and will be added to WPF is the extremely cool Visual State Manager (VSM) for controls. VSM simplifies the development of interactive control templates by introducing visual states and state transitions. Since the introduction of WPF, developers could customize the look, feel, and behavior of WPF controls through templates and styles. For instance, you could not only change the shape and appearance of a control but also define a new action or animation for when the control is clicked, gets or loses focus, and so on.
With VSM, visual states and state transitions do some of that work for you. Typical VSM visual states are normal, mouse-over, disabled, and focused. For each of these states you can define a style or a shape. State transitions define how the control moves from one state to another. You typically define transitions through animations. At run time, Silverlight will play the appropriate animation and apply the specified style to elegantly and smoothly move the control from state to state.
To define a visual state, insert a fragment of markup in the control template, as shown here:
Copy Code
```
<vsm:VisualStateManager.VisualStateGroups>
  <vsm:VisualStateGroup x:Name="CommonStates">
    <vsm:VisualState x:Name="MouseOver">
      <Storyboard>
        ...
      </Storyboard>
    </vsm:VisualState>
  </vsm:VisualStateGroup>
</vsm:VisualStateManager.VisualStateGroups>
```
Each control defines one or more groups of states such as CommonStates and FocusStates. Each group, in turn, defines specific visual states such as MouseOver, Pressed, and Checked. For each visual state and transition between states, you can define a storyboard that Silverlight will automatically play when appropriate.
In a nutshell, there are features in WPF that Silverlight 2 doesn't support and features in Silverlight 2 that WPF lacks. These differences affect mostly the compatibility at the XAML level. For complete compatibility, you can use the common subset, which, thankfully, is large enough to still let you do virtually anything.

Sharing Code

When VSM becomes available for the desktop version of WPF, you'll be able to use the same approach for both Silverlight and Windows desktop applications and share control templates between WPF and Silverlight projects. Until then, let's see how you can reuse code between WPF desktop and Web projects today.
A WPF app is made of XAML and managed code. The managed code targets the classes of the supported version of .NET. It is your responsibility to use a common subset of XAML that both desktop WPF and Silverlight 2 can understand. Likewise, it is your responsibility to organize your codebehind classes such that differences in the back-end framework are handled properly.
There are basically two scenarios in which you'll want to reuse WPF code. One is where you have an existing WPF desktop application and want to offer it over the Web to simplify maintenance and deployment. The other is when you want to develop a front end for an existing system and offer it for both Windows and Web clients.
I'll approach the question of reusing code from the WPF-to-Silverlight perspective. In terms of patterns to refactor code and markup, there's very little difference.

Reasoning about WPF Applications

A typical WPF application is built from a tree of objects where Window is the root of the tree. In turn, the Window element contains a number of child elements laid out or stacked in a variety of ways. Elements refer to basic shapes, layout managers, storyboards, and controls (including custom third-party and user controls). Here's a basic example:

Copy Code

```
<Window x:Class="Samples.MyTestWindow"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="Test App" Height="300" Width="350">
    <StackPanel Margin="10">
        ...
    </StackPanel>
</Window>
```

This code cannot be incorporated into a Silverlight 2 application as is. First of all, the Window element is not supported in Silverlight. In the Silverlight assemblies, there is simply no such class in the System.Windows namespace. The root tag of any Silverlight application is UserControl. The element is mapped to the UserControl class defined in the System.Windows.Controls namespace.

Here's the modified heading of a Silverlight application:

Copy Code

```
<UserControl x:Class="Samples.MyTestWindow"
    xmlns="http://schemas.microsoft.com/client/2007"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="Test App" Height="300" Width="350">
    <StackPanel Margin="10">
        ...
    </StackPanel>
</UserControl>
```

All the markup code you have within the boundaries of the <Window> and <UserControl> elements should continue to match. It is up to you to modify the original desktop XAML to make it match the requirements of the Silverlight XAML parser. This task largely benefits from the high level of compatibility between the two syntaxes. Most of the time, adjusting XAML is a matter of fixing a few elements and attributes. Let me provide a couple of examples.

Desktop WPF features a <label> element that Silverlight doesn't recognize. When porting this markup to Silverlight, you have to find a workaround for the label. A rectangle with a text block inside may be a workable solution.

In WPF, you can associate a tooltip with controls using the ToolTip attribute, as you see in the following:

Copy Code

```
<Button Tooltip="Click me" ... />
```

In Silverlight 2, the ToolTip attribute is not supported. You have to resort to the ToolTipService, as in the following code snippet:

Copy Code

```
<Button ToolTipService.ToolTip="Click me" ... />
```

It should be noted that both solutions work in desktop WPF. In addition, the ToolTipService offers a number of additional tooltip properties in desktop WPF such as placement, offset, initial delay, and duration. However, none of these extra properties are supported by Silverlight.

Are these all of the compatibility issues between WPF and Silverlight? It depends on how you use WPF. In general, porting a nontrivial WPF application to Silverlight is difficult and not really even a primary goal. To begin with, in Silverlight 2 you do not have triggers everywhere you might need them. For example, there is a Triggers collection on UI elements—descendants of FrameworkElement—but not in styles, data, and control templates.

Likewise, data binding is supported in Silverlight but not in the same manner as it is in WPF. For example, you have the Binding element, you have data context, data templates, and observable collections. As mentioned, you have no triggers and a simplified XAML markup that requires you to write code more often

than in WPF. Plus, the internal implementation is quite different. The Binding object in Silverlight has much fewer properties than in WPF.
Globalization is another area that might cause you headaches. For performance reasons, the core CLR doesn't include its own globalization data for all supported cultures. Instead, the CultureInfo class in Silverlight relies on the globalization functionality provided by the underlying operating system. This means that there's no way for you to give applications the same globalization settings across different operating systems.
Finally, WPF has a richer set of controls that are not available in Silverlight. A good example is the RichTextBox control.
In summary, porting a Silverlight application to WPF is relatively trivial, although as a developer you should be concerned with possible performance problems that could be caused by the richer object model. The key fact to remember is that Silverlight supports a subset of the possibilities offered by desktop WPF; it is up to you to devise a cross-platform solution by choosing the appropriate features.

Writing Cross-Platform WPF Code

The easiest way to port code from WPF projects to Silverlight projects is through user controls. Aside from a distinct XML namespace and markup differences, a user control is the only way to share markup and code between desktop and Web-based WPF.
If your WPF application is natively organized, or can be refactored, to make extensive use of user controls, porting code to Silverlight will be much easier than the other, less appealing, option of cutting and pasting.
So can assemblies be shared between WPF and Silverlight projects? In Silverlight 2, you can certainly create custom class libraries packed into assemblies. However, you should be aware that these libraries target the Silverlight version of the .NET Framework and use a different security model (for more see this month's installment of CLR Inside Out).
Any assemblies you use in a desktop WPF application must be recompiled as Silverlight class libraries to ensure that they refer to the correct assemblies and make legal calls to classes. Needless to say, this process of recompiling may generate additional work to fix calls to unsupported classes.
In summary, with a bit of work you can certainly take a WPF application and expose it over the Web through Silverlight. And when you do so, you are actually exposing your code over a number of non-Windows platforms without forcing clients to have the full .NET Framework installed. **Figure 1** shows a simple WPF standalone desktop application. **Figure 2** shows the same application hosted in Internet Explorer through Silverlight.

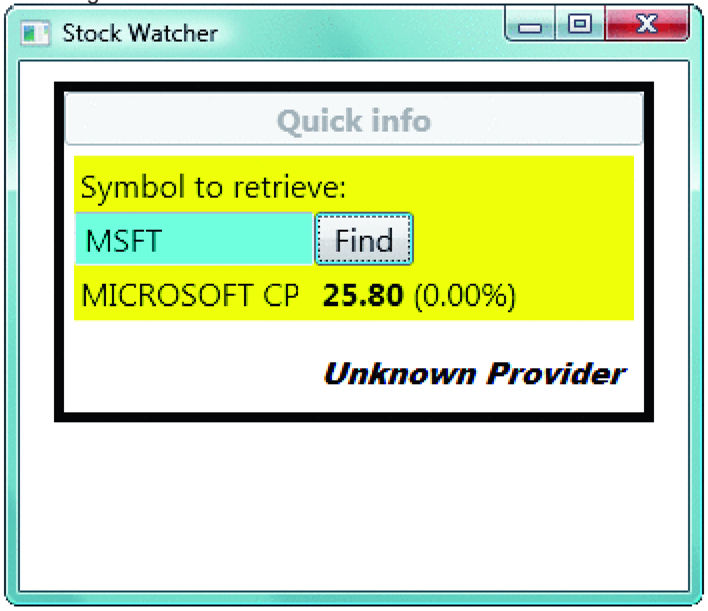
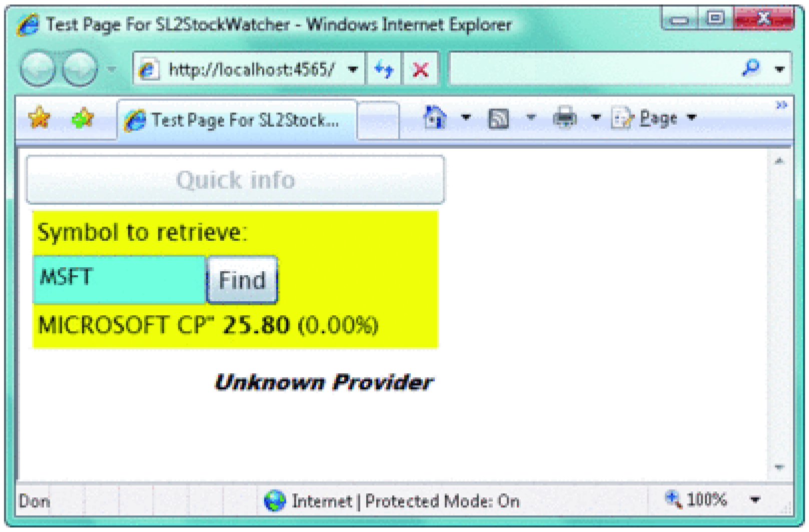


Figure 1 **A Sample WPF Application**


Test Page For SL2StockWatcher - Windows Internet Explorer
http://localhost:4565/
Test Page For SL2Stock...
Page
Quick info
Symbol to retrieve:
MSFT
Find
MICROSOFT CP" 25.80 (0.00%)
Unknown Provider
Don
Internet | Protected Mode: On
100%
Figure 2 **A WPF Application Adapted for Silverlight** (Click the image for a larger view)

Analyzing Managed Code

For some reason, code reuse between WPF and Silverlight is perceived to be a XAML problem. As mentioned, you do have some XAML issues when adapting your code, but the biggest challenge is the codebehind class. In both Windows and Silverlight, a XAML file is paired with a codebehind class written in C# or another managed language. Unfortunately, these codebehind classes target different versions of the .NET Framework. The desktop version of WPF relies on the full base class library (BCL) of the .NET Framework 3.5, while Silverlight 2 uses a lightweight version of the BCL.

The Silverlight version of the BCL is significantly smaller, but it still supports fundamental capabilities such as collections, reflection, regular expressions, string manipulation, threading, and timers. You also have tools to call into a variety of services such as XML Web services, WCF services, and ADO.NET Data Services. In addition, you have a rich networking support to communicate over HTTP—with Plain Old XML (POX) and Representational State Transfer (REST) services—and, in general, reach any public HTTP endpoint. Networking support also includes (cross-domain) sockets and duplex communication.

Finally, the Silverlight BCL provides good support for working with XML data, including ad hoc versions of the XmlReader and XmlWriter classes. These classes are quite similar to the analogous classes in the desktop version of the .NET Framework.

Building on these core capabilities, in Silverlight 2, you find full support for LINQ to Objects, LINQ to XML, and expression trees. Starting with Beta 2, Microsoft also added a brand new LINQ to JavaScript Object Notation (JSON) provider to run LINQ queries directly on JSON data.

Another point to mention is that networking in Silverlight can only happen asynchronously. To make a synchronous call you have to resort to a trick that essentially consists of invoking the browser interoperability layer and reaching the browser's implementation of XMLHttpRequest, whenever this is possible (for details on this, see go.microsoft.com/fwlink/?LinkId=124048).

The bottom line is that WPF and Silverlight codebehind classes leverage significantly different class libraries. This fact, much more than any XAML differences, hinders reusability. Let's tackle this problem next.

Get a Strategy for Critical Code

When you write code to be shared by Silverlight and Windows runtimes, you should know very well what each platform supports. Ideally, you'll have a list of requested tasks that require special handling on each platform. Next, you isolate these pieces of code in an object and extract an interface out of them.

As a result, you'll have a fully reusable block of code that calls into separate components for critical functionalities. Porting such an application to Silverlight (or WPF) would then be as easy as replacing critical components with others that are platform specific. Because all these components still expose a common interface, their implementation is transparent to the main block of code.

The pattern behind this approach is the "Strategy" pattern. For a formal definition of this, you should see go.microsoft.com/fwlink/?LinkId=124047. In a nutshell, the Strategy pattern is helpful whenever you need to dynamically change the algorithm used to perform a certain task in an application.
Once you have identified a region of your code that has the potential to vary based on runtime conditions, you define an interface that specifies the abstract behavior of your code. Next, you create one or more strategy classes that implement the interface and thereby represent different ways in which the abstract behavior can be accomplished. By then changing the strategy class, you are simply changing your approach for solving the algorithm defined by the interface. In doing so, you decouple the behavior's actual implementation (the strategy) from the code that is using it.
In ASP.NET, for example, the Strategy pattern is used in the implementation of the provider model for membership, roles, user profiles, and so forth. The ASP.NET runtime knows that there's a particular interface to deal with, say, membership and users. It also knows how to find and instantiate the concrete classes for these interface types. However, the runtime components depend only on the interface, making the details of the concrete class irrelevant to ASP.NET.

Going through an Example

Let's consider the sample application shown inFigures **1** and **2**. In both Silverlight and Windows, the application presents a form where users can type a stock symbol to get a current quote. **Figure 3** shows the diagram of the application when the user clicks the button. The user interface employs a Model-View-Presenter (MVP) pattern to convey in a single presenter class all the logic behind the UI. The presenter, in turn, invokes an internal QuoteService class that ultimately responds by providing a StockInfo object with any information to be incorporated in the user interface (see **Figure 4**).

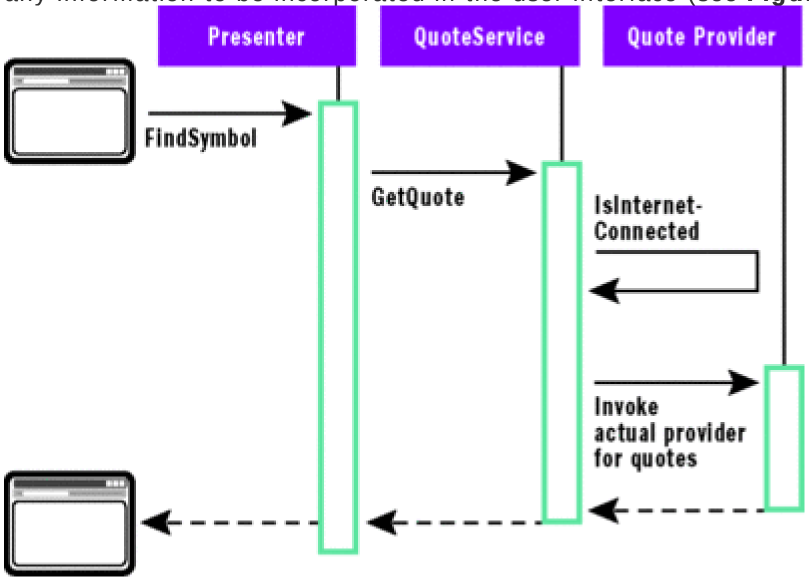


Figure 3 **Diagram of the Application's Behavior** (Click the image for a larger view)

Figure 4 The Presenter Class

Copy Code

```
namespace Samples
{
    class SymbolFinderPresenter
    {
        // Internal reference to the view to update
        private ISymbolFinderView _view;

        public SymbolFinderPresenter(ISymbolFinderView view) {
            this._view = view;
        }

        public void Initialize() { }
        // Triggered by the user's clicking on button Find
        public void FindSymbol() {
            // Clear the view before operations start
            ClearView();

            // Get the symbol to retrieve
            string symbol = this._view.SymbolName;
            if (String.IsNullOrEmpty(symbol)) {
                _view.QuickInfoErrorMessage = "Symbol not found.";
                return;
            }

            QuoteService service = new QuoteService();
            StockInfo stock = service.GetQuote(symbol);

            // Update the view
            UpdateView(stock);
        }

         private void ClearView() {
            _view.SymbolDisplayName = String.Empty;
            _view.SymbolValue = String.Empty;
            _view.SymbolChange = String.Empty;
            _view.ServiceProviderName = String.Empty;
        }

        private void UpdateView(StockInfo stock) {
             // Update the view
            _view.QuickInfoErrorMessage = String.Empty;
            _view.SymbolDisplayName = stock.Symbol;
            _view.SymbolValue = stock.Quote;
            _view.SymbolChange = stock.Change;
            _view.ServiceProviderName = stock.ProviderName;
        }
     }
}
```

The QuoteService class doesn't attempt to retrieve quotes itself. It first creates a provider component and then yields to it. The QuoteService class implements a very simple algorithm. If there's an Internet connection, it uses a provider class that uses a public Web service to get financial data; otherwise, it switches to a fake provider that just returns random numbers. So at some point, the QuoteService class needs to check for an Internet connection, as shown in **Figure 5**.

Figure 5 QuoteService Checks for Internet Connection

Copy Code

```
public StockInfo GetQuote(string symbol)
{
    // Get the provider of the service
    IQuoteServiceProvider provider = ResolveProvider();
    return provider.GetQuote(symbol);
}
private IQuoteServiceProvider ResolveProvider()
{
    bool isOnline = IsConnectedToInternet();
    if (isOnline)
        return new FinanceInfoProvider();
    return new RandomProvider();
}
```

So far, there's no difference between Silverlight and WPF, and all the code is fully reusable. To check for an Internet connection in .NET, you can use some of the static methods on the NetworkInterface object. The object is defined in the System.Net.NetworkInformation namespace. In particular, the GetIsNetworkAvailable method returns a Boolean value that denotes whether any network connection is available. Unfortunately, this doesn't say much about Internet connectivity. To make sure you also have Internet access, the only reliable method is to attempt to ping a host (see **Figure 6**).

Figure 6 Ping for Internet Connection

Copy Code

```
private bool IsConnectedToInternet()
{
    string host = "...";
    bool result = false;
    Ping p = new Ping();
    try
    {
        PingReply reply = p.Send(host, 3000);
        if (reply.Status == IPStatus.Success)
            return true;
    }
    catch { }

    return result;
}
```

The only problem in **Figure 6** is that the code isn't supported in Silverlight 2. (And likewise, it isn't supported by the NetworkInterface object.) You need to isolate this code (and any other code you spot with potential compatibility problems) in a replaceable class. (For more on this restriction, see the CoreCLR In Silverlight sidebar accompanying this column.) In the companion source code, I create a utility interface for these kinds of potentially problematic methods and then create strategy classes that implement the interface for each platform, as shown in **Figure 7**.

Figure 7 Component for Checking Internet Connection

Copy Code

```
public partial class SilverCompatLayer : ICompatLib
{
    public bool IsConnectedToInternet()
    {
        string host = "...";
        bool result = false;
        Ping p = new Ping();
        try
        {
            PingReply reply = p.Send(host, 3000);
            if (reply.Status == IPStatus.Success)
                return true;
        }
        catch { }

        return result;
    }

    public string GetRawQuoteInfo(string symbol)
    {
        string output = String.Empty;
        StreamReader reader = null;
        // Set the URL to invoke
        string url = String.Format(UrlBase, symbol);

        // Connect and get response
        WebRequest request = WebRequest.Create(url);
        WebResponse response = request.GetResponse();

        // Read the response
        using (reader = new StreamReader(response.GetResponseStream()))
        {
           output = reader.ReadToEnd();
           reader.Close();
        }

        return output;
    }

    // A few other methods that require a different implementation
    // (See source code)
    ...
}
```

The interface here decouples the platform-specific strategy class from the hosting application. By then hiding the code that instantiates the concrete strategy class behind a factory method, the following code can be used safely in both WPF and Silverlight:

Copy Code

```
private bool IsConnectedToInternet()
{
    ICompatLib layer = ServiceResolver.ResolveCompatLayer();
    return layer.IsConnectedToInternet();
}
```

The SilverCompatLayer class lives in a separate assembly. This assembly is all that you have to change when porting your WPF code to Silverlight, or vice versa.

After creating the necessary platform-specific strategy classes, all that remains is to create a Silverlight version of the application. The Silverlight project that is derived from the original WPF application contains exact copies of all files except for the compatibility assembly.

You'll notice in the code download associated with this article that I determined which strategy implementation to use by instantiating the concrete class explicitly in a factory method. You can instantiate classes directly like this or read their names from a configuration file and use reflection to get instances. These are implementation details that mostly depend on the type of system you're building. As far as the WPF-to-Silverlight compatibility is concerned, strategies and layering are the key concepts to understand.

Final Considerations

In the sample application, I make a network call. In the original WPF application, the call is synchronous. In Silverlight 2, however, there's no support at all for synchronous network calls. The only way you can make synchronous calls is by using the aforementioned trick based on calling the browser's implementation of XMLHttpRequest. (See the source code for details.)

In the sample code, I've successfully ported my original WPF application to the Web. When porting your code, you might also want to consider the native features of the Silverlight environment and modify the structure of your application when fitting. Note that in my simple example, rewriting the application using the Silverlight programming model would have taken less code and less effort than adapting it from a WPF application.

So WPF and Silverlight have a lot in common when it comes to purely visual things such as XAML. The underlying programming model, however, is a bit different, and solutions that work in WPF cannot always be applied as is to Silverlight and vice versa. This said, sharing XAML and code between WPF and Silverlight applications can definitely be done.

Send your questions and comments for Dino to cutting@microsoft.com.

Dino Esposito is currently an architect at IDesign and the author of *Programming ASP.NET 3.5 Core References*. Based in Italy, Dino is a frequent speaker at industry events worldwide. You can join his blog at weblogs.asp.net/despos