MAY 2013 VOLUME 28 NUMBER 05

MVVM - Commands, RelayCommands and EventToCommand

By Laurent Bugnion | May 2013

In previous installments of this series, I described the importance of decoupling the components of an application to make the application easier to unit test, maintain and extend. I also showed how adding design-time data makes it easier to work in Expression Blend or the Visual Studio designer in a visual manner.

In this article, I take a closer look at one of the important components of any Model-View-ViewModel application: the command. Historically, the .NET Framework has always been an event-based framework: a class exposes an event that is raised by the class instances when subscribers need to be notified. On the other hand, the subscribers provide an EventHandler, which is typically a method with two parameters: the sender of the event and an instance of a class deriving from EventArgs. When the event is raised, the event-handling method is executed and the EventArgs instance carries additional information (if available) about what caused the event in the first place.

This approach is fairly simple and successful for many scenarios. In .NET, it is often used to call back a subscriber after a Web operation completes (or fails). It is used by a sensor (such as location, orientation, proximity and so on) to notify the class that uses it that a condition has changed (for example, the user has moved, the screen has rotated, the device is close to another one, and the like). Most notably, this approach is used by UI elements to handle user events—for example, the click of a button, the movement of the mouse and many more.

For all their utility, event handlers have one problematic side effect: they can create a tight coupling between the instance that exposes the event and the instance that subscribes to it. The system needs to keep track of event handlers so that they can be executed when the event is raised, but the strong link this creates might prevent garbage collection. Of course, this isn't an issue if the event handler is a static method, but it is not always possible to handle all events with static methods only. This is a frequent cause for memory leaks in .NET.

Another consequence of the tight coupling between an event and its handler is that the event handler for a UI element declared in XAML must be found in the attached code-behind file. If it is not there (or if there is no attached code-behind file), the compilation will fail with an error. This is especially an issue when working with list controls and associated DataTemplates. When an element of the template must be actuated, an event handler can be defined, but as a consequence, the DataTemplate cannot be moved into an external ResourceDictionary, as shown in **Figure 1**. This code would cause a compilation error.

Figure 1. DataTemplate with Event Handler

```
<!--In MainPage.xaml-->
<GridView
  ItemsSource="{Binding DataItems}"
  ItemTemplate="{StaticResource DataItemTemplate}" />
<!--In an external resource dictionary-->
<ResourceDictionary
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">

  <DataTemplate x:Key="DataItemTemplate">
    <!--This causes a compilation error-->
    <StackPanel Tapped="ItemTapped">
      <!--...-->
    </StackPanel>
  </DataTemplate>
</ResourceDictionary>
// In MainPage.xaml.cs
private void ItemTapped(
  object sender,
  TappedRoutedEventArgs e)
{
  var panel = (FrameworkElement)sender;
  var item = (DataItem)panel.DataContext;
  ((Frame)Window.Current.Content).Navigate(typeof (DetailsPage), item);
}
```

Thankfully, there is a solution to this issue: using a command to expose the “event handler” and bind the UI element to that command by using a XAML data-binding. Because data-bindings are evaluated only at run time, they won’t cause a compilation error. And because they are loosely coupled, they won’t risk causing memory leaks.

What’s a Command?

Commands are an implementation of the ICommand interface that is part of the .NET Framework. This interface is used a lot in MVVM applications, but it is useful not only in XAML-based apps. The ICommand interface specifies three members:

- The method Execute(object) is called when the command is actuated. It has one parameter, which can be used to pass additional information from the caller to the command.
- The method CanExecute(object) returns a Boolean. If the return value is true, it means that the command can be executed. The parameter is the same one as for the Execute method. When used in XAML controls that support the Command property, the control will be automatically disabled if CanExecute returns false.

- The CanExecuteChanged event handler must be raised by the command implementation when the CanExecute method needs to be reevaluated. In XAML, when an instance of ICommand is bound to a control's Command property through a data-binding, raising the CanExecuteChanged event will automatically call the CanExecute method, and the control will be enabled or disabled accordingly.

Note that in Windows Presentation Foundation (WPF), the CanExecuteChanged event does not need to be raised manually. A class named CommandManager is observing the user interface and calls the CanExecute method when it deems it necessary. In all other XAML frameworks, however (including Windows RT), the developer must take care of raising this event when it's needed.

Of course, having to implement the ICommand interface every time a command must be added to the project is impractical. This is why some of the most popular frameworks and toolkits in .NET offer a generic implementation of ICommand.

The RelayCommand

In the MVVM Light Toolkit, the open-source toolkit described in the previous articles in this series, the ICommand implementation is called RelayCommand. The constructor of this class has two parameters:

- The first parameter is compulsory. It is used for the Execute method that the ICommand interface requires. For example, a lambda expression can be used as shown in **Figure 3**. Alternatively, the syntax shown in **Figure 2** can be used, where a delegate to a method is provided for the Execute parameter.
- The second parameter is optional. It is a delegate for the CanExecute method that's specified by ICommand. This delegate must return a Boolean. Here, too, a lambda expression can be used, as shown in **Figure 2**, as can a delegate to a method defined somewhere else.

Figure 2. Creating a RelayCommand

```
public RelayCommand MyCommand
{
  get;
  private set;
}

public MainViewModel()
{
  MyCommand = new RelayCommand(
    ExecuteMyCommand,
    () => _canExecuteMyCommand);
}

private void ExecuteMyCommand()
{
  // Do something
}
```

The code in **Figure 3** shows a more compact syntax for a RelayCommand, accepting a parameter of type RssArticle. This example is taken from the RssReader sample application that was implemented in the previous articles.

The code in the property getter checks first (thanks to the "??" operator) whether the _navigateToArticleCommand attribute is already created. If yes, the attribute is returned. If not, the command is created and stored in the attribute before it is returned. The command takes one parameter of type RssArticle, as specified by the generic parameter in the command declaration. In XAML, as we will see later in **Figure 9**, the parameter can be specified with any XAML expression, such as a Binding, a StaticResource or an explicit value; in that last case (for example "True", "1234.45", or "Hello world"), the RelayCommand will attempt to convert the value (specified as a string in the XAML markup) to the type specified in the RelayCommand declaration. This conversion might fail if no converter is found in the system by the XAML parser or if the value is invalid.

Figure 3. The NavigateToArticleCommand (with One Parameter)

```
private RelayCommand<RssArticle> _navigateToArticleCommand;
public RelayCommand<RssArticle> NavigateToArticleCommand
{
  get
  {
    return _navigateToArticleCommand
      ?? (_navigateToArticleCommand = new RelayCommand<RssArticle>(
        article =>
        {
          _navigationService.NavigateTo(typeof (DetailsPage), article);
        }));
  }
}
```

Figure 4 shows the RefreshCommand, which is used in the RssReader sample application to reload the list of articles from the server. This command doesn't require any parameter.

Figure 4. The RefreshCommand (No Parameters)

```
private RelayCommand _refreshCommand;

public RelayCommand RefreshCommand
{
  get
  {
    return _refreshCommand
      ?? (_refreshCommand = new RelayCommand(
        async () =>
        {
          await Refresh();
        }));
  }
}
```

A developer can specify when the command can be executed by implementing the CanExecute delegate as the second parameter of the RelayCommand constructor. For instance, the Refresh method should not be called while the Web client is waiting for a response from the Web server. The easiest way to do that is to disable the control used to execute the command. However, at the time when the RefreshCommand is implemented, the developer does not know what kind of control will be used by the designer. By specifying the CanExecute delegate (and calling the CanExecuteChanged event at appropriate times), the developer can reach the desired effect without having to worry about impacting the UI. To do that, we can modify the RefreshCommand as shown in **Figure 5**. When RefreshCommand.RaiseCanExecuteChanged() is called, the value if _isRefreshing is evaluated and the bound button will be disabled or enabled accordingly.

Figure 5. Implementing the CanExecute Delegate in the RefreshCommand

```
private bool _isRefreshing;
private RelayCommand _refreshCommand;

public RelayCommand RefreshCommand
{
  get
  {
    return _refreshCommand
      ?? (_refreshCommand = new RelayCommand(
        async () =>
        {
          if (_isRefreshing)
          {
            return;
          }

          _isRefreshing = true;
          RefreshCommand.RaiseCanExecuteChanged();

          await Refresh();

          _isRefreshing = false;
          RefreshCommand.RaiseCanExecuteChanged();
        },
        () => !_isRefreshing));
  }
}
```

Using a Command in XAML

One advantage of using commands is that it encourages a clean workflow when developing the application. On one hand, developers concentrate on exposing functionality in the view models by creating properties of type RelayCommand and implementing the Execute and (optionally) the CanExecute delegates. On the other hand, whoever is in charge of the user interface implementation (sometimes a designer, sometimes an integrator) can decide which controls should be used to actuate the command. At a later time, the UI can be modified with different controls without having to change the command implementation in the view model. For instance, the RefreshCommand can be data-bound to a Button in a Windows 8 app's BottomApplicationBar with the markup shown in **Figure 6**. Because the command is exposed as a property on the MainViewModel, we can use a simple Binding syntax. And because the CanExecute delegate is implemented, clicking the button will cause the Refresh method to be called, and the Button will be disabled until the asynchronous operation is completed.

Figure 6. Binding the Button’s Command Property to the RefreshCommand in Windows 8

```
<Page.BottomAppBar>
  <AppBar>
    <Grid>
      <Grid.ColumnDefinitions>
        <ColumnDefinition Width="0.5*" />
        <ColumnDefinition Width="0.5*" />
      </Grid.ColumnDefinitions>

      <StackPanel Orientation="Horizontal">
        <Button Style="{StaticResource RefreshAppBarButtonStyle}"
                Command="{Binding RefreshCommand}" />
      </StackPanel>
    </Grid>
  </AppBar>
</Page.BottomAppBar>
```

In Windows Phone, one annoying limitation of the AppBar is that you cannot bind commands to an ApplicationBarIconButton. Libraries are available that offer a way around this limitation, such as the Bindable Application Bar or AppBarUtils referenced at the end of this article. However, you can overcome this limitation with just a few lines of code-behind, so it might not be necessary to use a third-party library. For example, in the Windows Phone version of the RssReader sample app, the RefreshCommand is executed with the code shown in **Figure** 7. Keep in mind that it is only the ApplicationBarIconButton that cannot be bound to a command. Normal Button instances in the UI have Command and CommandParameter properties just as in other XAML frameworks.

Figure 7. Executing the Command in Windows Phone

```
<!--In MainPage.xaml-->
<phone:PhoneApplicationPage.ApplicationBar>
  <shell:ApplicationBar>
    <shell:ApplicationBarIconButton
      IconUri="/Assets/AppBar/sync.png"
      Text="refresh"
      Click="RefreshClick" />
  </shell:ApplicationBar>
</phone:PhoneApplicationPage.ApplicationBar>

// In MainPage.xaml.cs
private void RefreshClick(object sender, EventArgs e)
{
  var vm = (MainViewModel)DataContext;
  vm.RefreshCommand.Execute(null);
}
```

Commands for Every Element

Unfortunately, only a rather small subset of controls support commands out of the box, with Command and CommandParameter properties. In addition, the bound command will be actuated only when the control is clicked (or tapped with a finger), but no other events are supported. In the case of the RssReader's NavigateToArticleCommand, we want it to be actuated when the item is tapped in the list control. We already saw that using a command instead of an event handler allows us flexibility in moving the DataTemplate to an external ResourceDictionary. Now we need the flexibility of invoking a command on any UI element (not just a button) for any event. This is where Blend behaviors come to the rescue. Note that at the time of writing, Blend behaviors are not available for Windows 8. They are, however, available for every other XAML-based framework. Later in this article, you'll see how to work around this limitation in Windows 8.

Blend behaviors are referred to this way because they were developed by the Blend team, but they do not require Blend. In fact, they are strongly inspired by a pattern called Attached Behaviors, built with attached properties in XAML. Attached behaviors were used by the XAML community before Blend behaviors appeared. (Two good articles about attached behaviors are listed at the end of this article.) However, Blend behaviors improve toolability tremendously. Even though you don't need Blend to get these behaviors to work, using Blend makes adding a behavior to a UI element and configuring it much easier. To use Blend behaviors, you must add System.Windows.Interactivity.dll to your app's references. This reference is already added in every MVVM Light application by default.

Adding the NavigateToArticleCommand to the RssArticle DataTemplate is quite easy with the following steps:

1. Open the RssReader application's MainPage.xaml in Blend. For this, you can open the RssReader solution in Blend directly, or you can right-click MainPage.xaml in Visual Studio's Solution Explorer and select Open in Blend. Make sure that you select the Windows Phone project's MainPage.xaml, and not the one for Windows 8.
2. In Blend, in the Objects and Timeline pane, right-click the LongListSelector and select Edit Additional Templates, Edit Item Template, Edit Current.
3. Next to the Projects tab, select Assets and then select the Behaviors category.
4. Drag an EventToCommand from the Assets pane onto the StackPanel at the root of the ItemTemplate in the Objects and Timeline pane, as shown in **Figure 8**. Because the StackPanel's Background is set to Transparent (and not to No Brush), the command will be invoked when the user taps anywhere on the item, which improves the user experience.
5. With the new EventToCommand element selected, open the Properties pane and check that TriggerType is set to EventTrigger. Note that you can also use EventToCommand with other triggers, such as DataTrigger.
6. Set the EventName to Tap. We want the command to be invoked when the user taps the StackPanel.
7. In the Miscellaneous category, click the small square next to the Command property. This opens a context menu. Select Create Data Binding.

We want to create a binding between the Command property and the NavigateToArticleCommand located in the MainViewModel. Because this view model is exposed as the Main property of the ViewModelLocator, it is easy to locate in Blend's data-binding dialog.

1. In the Create Data Binding dialog, under Binding Type, select StaticResource. In MVVM Light applications, the ViewModelLocator is stored as a resource in App.xaml. This allows it to be found through any of Blend's data dialogs.

2. Under Resource, select Locator. Then, under Path, select Main/NavigateToArticleCommand and click OK.

The command needs a parameter of type RssArticle. We want to pass the current article, which is conveniently the DataContext of the ItemTemplate we are editing. Binding to the current DataContext is expressed by "empty binding": "{Binding}". To create such a binding in Blend, open the data-binding editor by clicking the small square next to the CommandParameter property and selecting Create Data Binding from the context menu. In the dialog, check the Custom check box above the Path field, and then click OK. This creates a binding with an empty Path, which is what we wanted in the first place.

Note: You can also use an InvokeCommandAction instead of MVVM Light's EventToCommand. This behavior is part of the System.Windows.Interactivity DLL. It is almost equivalent to EventToCommand, but without some of the advanced features.

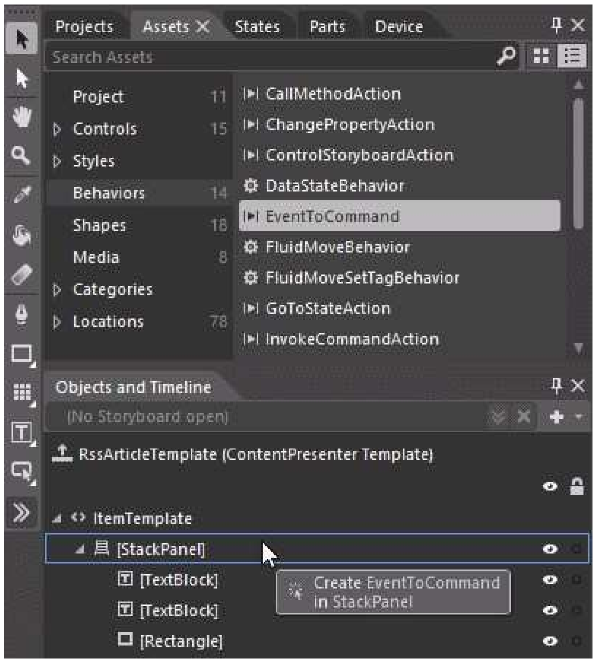

Figure 8. Adding the EventToCommand behavior in Blend

The XAML markup generated by this process is shown in **Figure 9**.

Figure 9. XAML Markup for EventToCommand

```
<phone:PhoneApplicationPage.Resources>
  <DataTemplate x:Key="RssArticleTemplate">
    <StackPanel Background="Transparent">
      <i:Interaction.Triggers>
        <i:EventTrigger EventName="Tap">
          <command:EventToCommand
            Command="{Binding Main.NavigateToArticleCommand,
              Mode=OneWay,
              Source={StaticResource Locator}}"
            CommandParameter="{Binding Mode=OneWay}" />
        </i:EventTrigger>
      </i:Interaction.Triggers>
      <!--...-->
    </StackPanel>
  </DataTemplate>
</phone:PhoneApplicationPage.Resources>
```

Using EventToCommand in Windows 8

In Windows 8, Blend behaviors are missing, as I mentioned earlier. In the RssReader application, open the Windows 8 project's MainPage.xaml. The lack of behaviors is solved here by handling the Tapped event on the DataTemplate's main Border and having the event handler in the MainPage.xaml.cs code-behind. Having to do this is annoying because it prevents us from moving the DataTemplate to a ResourceDictionary to have a cleaner XAML file.

To solve this issue, we can use a less toolable version of EventToCommand that's implemented using the Attached Behavior pattern mentioned earlier. You will find a Windows 8 implementation of EventToCommand in the Data folder of the sample application source code. Add this file to the Windows 8 version of the RssReader project—for example, to the Helpers folder. Then edit the RssArticleTemplate markup in MainPage.xaml as shown in **Figure 10**.

Figure 10. EventToCommand Behavior in Windows 8

```
<DataTemplate x:Key="RssArticleTemplate">
  <Border Height="140"
    Width="290"
    Padding="5"
    Background="Black"
    xmlns:b="using:Win8Utils.Behaviors"
    b:EventToCommand.Event="Tapped"
    b:EventToCommand.Command="{Binding Main.NavigateToArticleCommand,
      Mode=OneWay,
      Source={StaticResourceLocator}}"
    b:EventToCommand.CommandParameter="{Binding}">
    <!--...-->
  </Border>
</DataTemplate>
```

Of course, this version of EventToCommand has a few disadvantages. Adding new events requires the source code to be extended, but it is fairly easy based on the current implementation, which handles the Tapped, TextChanged and SelectionChanged events. Also, this implementation does not support tooling, which the Blend behaviors we used for the Windows Phone project do. It does, however, work properly and allows the DataTemplate to be moved in a ResourceDictionary, which was the goal.

Wrapping Up

This article has presented in-depth an important component of Model-View-ViewModel applications (which can also be used in non-MVVM apps): the ICommand interface and its RelayCommand implementation available in the MVVM Light Toolkit. You've seen how to use the EventToCommand behavior in Blend in Windows Phone and how to use an alternate version in Windows 8, which does not support Blend behaviors.

References

- The MVVM Light Toolkit can be downloaded at http://mvvmlight.codeplex.com.
- Bindable Application Bar for Windows Phone on Codeplex: http://bindableapplicationb.codeplex.com
- AppBarUtils for Windows Phone on Codeplex: http://appbarutils.codeplex.com
- "Introduction to Attached Behaviors in WPF" by Josh Smith: http://www.codeproject.com/Articles/28959/Introduction-to-Attached-Behaviors-in-WPF
- "The Attached Behavior Pattern" by John Gossman: http://blogs.msdn.com/b/johngossman/archive/2008/05/07/the-attached-behavior-pattern.aspx

---

Laurent Bugnion *is senior director for IdentityMine Inc., a Microsoft partner working with technologies such as Windows Presentation Foundation, Silverlight, Pixelsense, Kinect, Windows 8, Windows Phone and UX. He's based in Zurich, Switzerland. He is also a Microsoft MVP and a Microsoft Regional Director.*