



Articles » Platforms, Frameworks & Libraries » Windows Presentation Foundation » General

# Using RoutedCommands with a ViewModel in WPF



Josh Smith, 24 Jul 2008

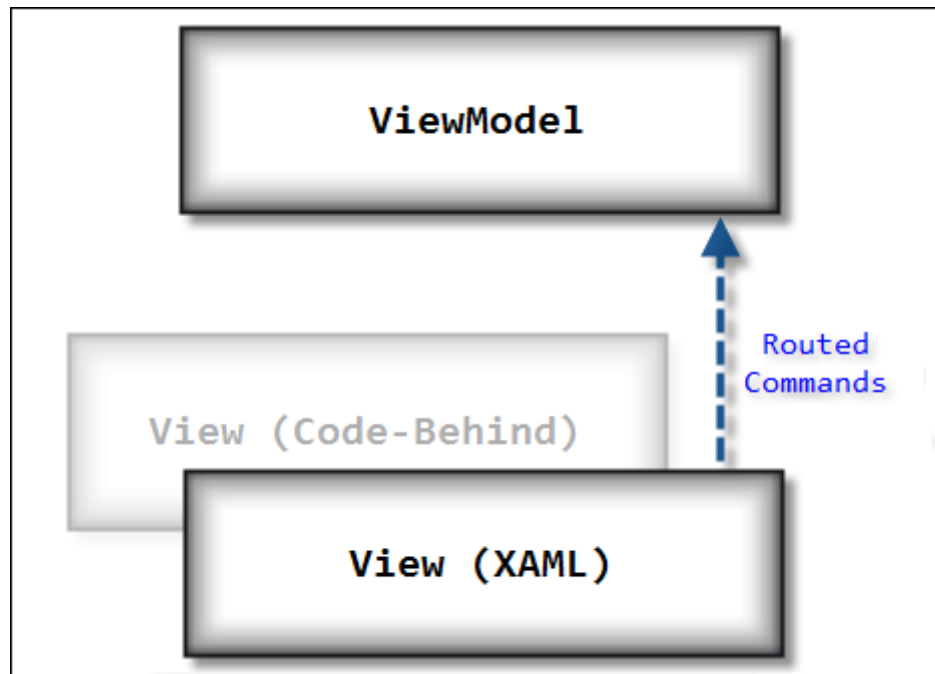
CPOL



4.96 (54 votes)

Reviews a clean and lightweight way to use RoutedCommands in the MVVM pattern.

**Download demo project (requires Visual Studio 2008) - 15.9 KB**



## Introduction

This article examines a technique that makes it easy to use routed commands when the command execution logic resides in ViewModel objects. This approach cuts out the middleman; the code-behind file. It allows your ViewModel objects to be the direct recipient of routed command execution and status query notifications.

This article assumes that the reader is already familiar with [data binding and templates](#), [routed commands](#), [attached properties](#), and the [Model-View-ViewModel \(MVVM\)](#) design pattern.

## Background

There was an [interesting thread](#) on the WPF Disciples forum about how to use routed commands in conjunction with the Model-View-ViewModel

pattern. After much discussion, and a side conversation with [Bill Kempf](#), I began to understand what the core issue was. Most examples of using WPF with the MVP, MVC, or MVVM patterns involve the use of routed commands. Those commands are accompanied by **CommandBindings** that point to event handling methods in the code-behind of the View, which, in turn, delegate to the Presenter/Controller/ViewModel associated with that View. The thread on the WPF Disciples forum revolved around a search to find a way for those **RoutedCommands** to talk directly to the ViewModel.

## The Benefits

There are several distinct benefits in having the routed commands in the View talk directly to the ViewModel. Bypassing the code-behind of the View means the View is that much less coupled to a ViewModel. It also means that the ViewModel is not dependent on the View's code-behind to properly handle a routed command's events and delegate those calls off to the correct members on the ViewModel objects. Not only that, but it reduces the amount of coding required to create a View, which is important when working in the Designer-Developer workflow.

## Prior Art

There are several existing solutions to this type of problem out there, from Dan Crevier's [CommandModel](#) to Rob Eisenberg's [Caliburn](#) framework. I felt that CommandModel was too complicated and restrictive, while Caliburn was too heavy and broad. I am not saying that they are in any way *bad* solutions, just not what I wanted for this particular task.

## My Solution

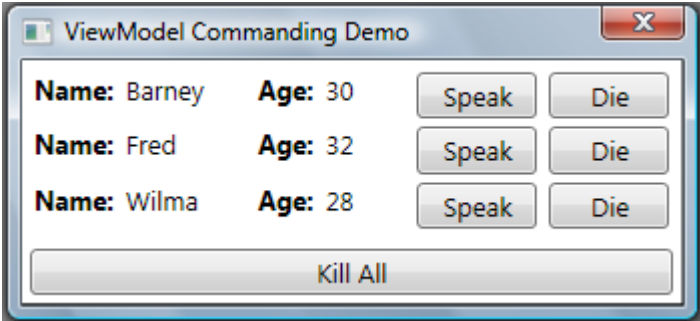
I thought long and hard about the issue, and decided that the solution is to create a custom **CommandBinding** class. To that end, I created the **CommandSinkBinding** class. It has two **CommandSink** properties: one is a normal instance property, the other is an attached property. The instance property provides a **CommandSinkBinding** with an object to handle its **CanExecute** and **Executed** events. The attached **CommandSink** property is used to specify the command sink to give to every **CommandSinkBinding** in an element's **CommandBindings** collection.

The other piece of the puzzle is a small interface called **ICommandSink** and a class that implements the interface, named **CommandSink**. The ViewModel classes that want to react to routed commands must implement that interface, or derive from **CommandSink**.

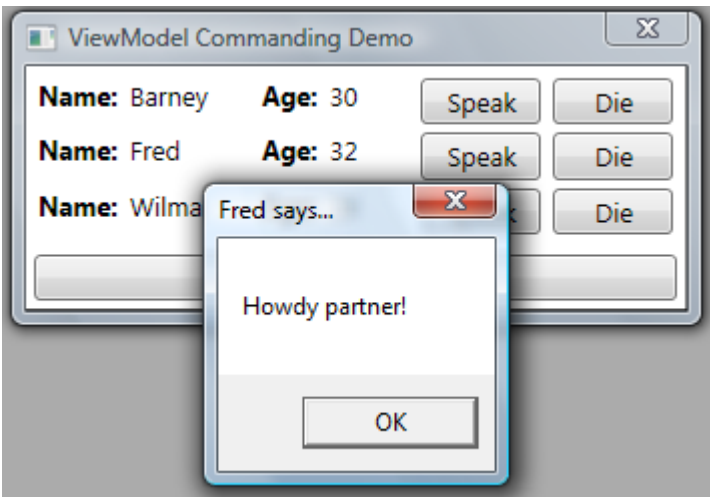
This solution is very lightweight, reusable, and does not rely on the use of Reflection at all.

## Seeing it in Action

Before we examine how my solution works, let's first take a look at the demo app, which is available for download at the top of this article. When you run the demo, it looks like this:



The UI shows three people, each of which you can tell to speak and die. If you click a person's Speak button, a message box pops up like so:



If you click a person's Die button, they become disabled and grey, as seen below:



The main **Window**'s code-behind file is unaltered. The content of its XAML file is listed below:

```
<Window
  x:Class="VMCommanding.DemoWindow"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:view="clr-namespace:VMCommanding.View"
  xmlns:vm="clr-namespace:VMCommanding.ViewModel"
  FontSize="13"
  ResizeMode="NoResize"
  SizeToContent="WidthAndHeight"
  Title="ViewModel Commanding Demo"
  WindowStartupLocation="CenterScreen"
>
  <Window.DataContext>
    <vm:CommunityViewModel />
  </Window.DataContext>

  <Window.Content>
    <view:CommunityView />
  </Window.Content>
</Window>
```

Clearly, there is not much going on in the **Window**. Let's now move our attention to the content of the **Window**: the **CommunityView** control. This user control knows how to render a **CommunityViewModel** object, which is basically just a container of **PersonViewModel** objects and a command that allows you to kill them all in one fell swoop.

We will examine the ViewModel classes later, but for now, here is the **CommunityView** control. It is important to note that the code-behind file of the **CommunityView** and **PersonView** controls have no code in them, aside from the auto-generated code that calls the standard **InitializeComponent**.

```
<UserControl
  x:Class="VMCommanding.View.CommunityView"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:jas="clr-namespace:VMCommanding"
  />
```

```

xmlns:view="clr-namespace:VMCommanding.View"
xmlns:vm="clr-namespace:VMCommanding.ViewModel"
jas:CommandSinkBinding.CommandSink="{Binding}"
>
<UserControl.CommandBindings>
  <jas:CommandSinkBinding
    Command="vm:CommunityViewModel.KillAllMembersCommand" />
</UserControl.CommandBindings>

<DockPanel Margin="4">
  <Button
    DockPanel.Dock="Bottom"
    Command="vm:CommunityViewModel.KillAllMembersCommand"
    Content="Kill All"
    Margin="0,8,0,0"
  />
  <ItemsControl ItemsSource="{Binding People}">
    <ItemsControl.ItemTemplate>
      <DataTemplate>
        <view:PersonView />
      </DataTemplate>
    </ItemsControl.ItemTemplate>
  </ItemsControl>
</DockPanel>
</UserControl>

```

The most relevant parts of that XAML is **bold**. As you can see, this control's **CommandBindings** contains a **CommandSinkBinding** whose **Command** property references a static command of the **CommunityViewModel** class. The Kill All button also references that command. Notice how the **UserControl** element has the **CommandSinkBinding**'s **CommandSink** attached property set to **{Binding}**. That means, it is bound to the **DataContext** of the **CommunityView** control, which is the **CommunityViewModel** object set on the main **Window**'s **DataContext**, as seen previously.

Each **PersonViewModel** in the community renders in an **ItemsControl**. That control's **ItemsSource** is bound to the **People** property of the **CommunityViewModel** object, which is the data context. Each item in the control is rendered by a **DataTemplate** which emits a **PersonView** control. Now, let's see **PersonView**'s XAML file:

```

<UserControl
  x:Class="VMCommanding.View.PersonView"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:jas="clr-namespace:VMCommanding"
  xmlns:vm="clr-namespace:VMCommanding.ViewModel"
  jas:CommandSinkBinding.CommandSink="{Binding}"
>
<UserControl.CommandBindings>
  <jas:CommandSinkBinding Command="vm:PersonViewModel.DieCommand" />
  <jas:CommandSinkBinding Command="vm:PersonViewModel.SpeakCommand" />
</UserControl.CommandBindings>

<UserControl.Resources>
  <Style TargetType="{x:Type TextBlock}">
    <Setter Property="Margin" Value="0,0,6,0" />
    <Style.Triggers>
      <DataTrigger Binding="{Binding CanDie}" Value="False">
        <Setter Property="Foreground" Value="#88000000" />
      </DataTrigger>
    </Style.Triggers>
  </Style>
</UserControl.Resources>

<StackPanel Margin="2" Orientation="Horizontal">
  <TextBlock Text="Name:" FontWeight="Bold" />
  <TextBlock Text="{Binding Name}" Width="60" />
  <TextBlock Text="Age:" FontWeight="Bold" />
  <TextBlock Text="{Binding Age}" Width="40" />
  <Button
    Command="vm:PersonViewModel.SpeakCommand"
    CommandParameter="Howdy partner!"
    Content="Speak"
    Margin="0,0,6,0"
  />
</StackPanel>

```

```

        Width="60"
    />
<Button
    Command="vm:PersonViewModel.DieCommand"
    Content="Die"
    Width="60"
/>
</StackPanel>
</UserControl>

```

Aside from having more stylistic resources and visual elements, this control is essentially the same as the **CommunityView** control. It, too, has **CommandSinkBindings** in its **CommandBindings** collection, it has the **CommandSink** attached property set on it, and the buttons it contains have their **Command** property set to static command fields of a **ViewModel** class. When the commands in this control execute, they will be handled by logic in the **PersonViewModel** class, instead of the **CommunityViewModel** class. That makes sense, considering this is a View of the **PersonViewModel** class.

Once again, it is important to note that the code-behind file of the **CommunityView** and **PersonView** controls have no code in them, aside from the auto-generated code that calls **InitializeComponent**. The main **Window**'s code-behind file is also unaltered. If you are accustomed to working with routed commands, this might seem both strange and wonderful at the same time!

## How it Works

Now it is time to turn our attention to how this works. You do not have to read this section of the article in order to use my solution, though I strongly recommend you do for the sake of understanding.

### ICommandSink

First, we will examine the **ICommandSink** interface. Remember, this is the interface that the **ViewModel** classes implement; it is implemented by the **CommandSink** class, and it is the type of the **CommandSink** property of **CommandSinkBinding**.

```

/// <summary>
/// Represents an object that is capable of being notified of
/// a routed command execution by a CommandSinkBinding. This
/// interface is intended to be implemented by a ViewModel class
/// that honors a set of routed commands.
/// </summary>
public interface ICommandSink
{
    bool CanExecuteCommand(ICommand command, object parameter, out bool handled);
    void ExecuteCommand(ICommand command, object parameter, out bool handled);
}

```

This is similar to the standard **ICommand** interface signature, only it is intended to be implemented by a class that can work with multiple commands. I included the **handled** parameter in both methods because I use it to set the **Handled** property of the event argument passed into a **CommandSinkBinding**'s event handling methods. Be sure to set that argument to **true** if your logic has successfully completed executing the command logic, or provide an execution status, because it helps improve performance in situations where there are large element trees and many routed commands.

### CommandSinkBinding

The real meat is in the **CommandSinkBinding** class. The class declaration and the instance **CommandSink** property is below:

```

/// <summary>
/// A CommandBinding subclass that will attach its
/// CanExecute and Executed events to the event handling
/// methods on the object referenced by its CommandSink property.
/// Set the attached CommandSink property on the element
/// whose CommandBindings collection contain CommandSinkBindings.
/// If you dynamically create an instance of this class and add it
/// to the CommandBindings of an element, you must explicitly set
/// its CommandSink property.
/// </summary>

```

```

public class CommandSinkBinding : CommandBinding
{
    ICommandSink _commandSink;

    public ICommandSink CommandSink
    {
        get { return _commandSink; }
        set
        {
            if (value == null)
                throw new ArgumentNullException("...");

            if (_commandSink != null)
                throw new InvalidOperationException("...");

            _commandSink = value;

            base.CanExecute += (s, e) =>
            {
                bool handled;
                e.CanExecute = _commandSink.CanExecuteCommand(
                    e.Command, e.Parameter, out handled);
                e.Handled = handled;
            };

            base.Executed += (s, e) =>
            {
                bool handled;
                _commandSink.ExecuteCommand(
                    e.Command, e.Parameter, out handled);
                e.Handled = handled;
            };
        }
    }

    // Other members omitted for clarity...
}

```

Now, we will turn our attention to the attached **CommandSink** property of **CommandSinkBinding**. This property enables us to provide multiple **CommandSinkBindings** with the same command sink.

```

public static ICommandSink GetCommandSink(DependencyObject obj)
{
    return (ICommandSink)obj.GetValue(CommandSinkProperty);
}

public static void SetCommandSink(DependencyObject obj, ICommandSink value)
{
    obj.SetValue(CommandSinkProperty, value);
}

public static readonly DependencyProperty CommandSinkProperty =
    DependencyProperty.RegisterAttached(
        "CommandSink",
        typeof(ICommandSink),
        typeof(CommandSinkBinding),
        new UIPropertyMetadata(null, OnCommandSinkChanged));

static void OnCommandSinkChanged(
    DependencyObject depObj, DependencyPropertyChangedEventArgs e)
{
    ICommandSink commandSink = e.NewValue as ICommandSink;

    if (!ConfigureDelayedProcessing(depObj, commandSink))
        ProcessCommandSinkChanged(depObj, commandSink);
}

// This method is necessary when the CommandSink attached property
// is set on an element in a template, or any other situation in
// which the element's CommandBindings have not yet had a chance to be
// created and added to its CommandBindings collection.

```

```

static bool ConfigureDelayedProcessing(DependencyObject depObj, ICommandSink sink)
{
    bool isDelayed = false;

    CommonElement elem = new CommonElement(depObj);
    if (elem.IsValid && !elem.IsLoaded)
    {
        RoutedEventHandler handler = null;
        handler = delegate
        {
            elem.Loaded -= handler;
            ProcessCommandSinkChanged(depObj, sink);
        };
        elem.Loaded += handler;
        isDelayed = true;
    }

    return isDelayed;
}

static void ProcessCommandSinkChanged(DependencyObject depObj, ICommandSink sink)
{
    CommandBindingCollection cmdBindings = GetCommandBindings(depObj);
    if (cmdBindings == null)
        throw new ArgumentException("...");

    foreach (CommandBinding cmdBinding in cmdBindings)
    {
        CommandSinkBinding csb = cmdBinding as CommandSinkBinding;
        if (csb != null && csb.CommandSink == null)
            csb.CommandSink = sink;
    }
}

static CommandBindingCollection GetCommandBindings(DependencyObject depObj)
{
    var elem = new CommonElement(depObj);
    return elem.IsValid ? elem.CommandBindings : null;
}

```

It is important to note that setting the attached **CommandSink** property only has an effect once for an element. You cannot set it again, nor set it to **null**. If you subsequently add a **CommandSinkBinding** to that element's **CommandBindings**, you will have to explicitly set that object's **CommandSink** property to an **ICommandSink** object. The **CommandBindingCollection** class does not provide collection change notifications, so I have no way of knowing if or when new items are added to the collection.

## CommandSink

Now that we have seen the fundamental pieces of the solution, let's turn our attention to a very convenient class called **CommandSink**, which implements the **ICommandSink** interface seen above. This class was suggested to me by the brilliant [Bill Kempf](#) shortly after I published this article. It provides a consolidated, reusable way to cleanly implement ViewModel objects that use this pattern. It can be used as a base class for your ViewModel objects, or, if you already have a base class for your ViewModel, you can embed an instance of **CommandSink** into a ViewModel class. I suggest you derive your ViewModel classes from **CommandSink** if possible, because it results in less code you have to write and maintain.

Here is the **CommandSink** class, in its entirety:

```

/// <summary>
/// This implementation of ICommandSink can serve as a base
/// class for a ViewModel or as an object embedded in a ViewModel.
/// It provides a means of registering commands and their callback
/// methods, and will invoke those callbacks upon request.
/// </summary>
public class CommandSink : ICommandSink
{
    #region Data
    readonly Dictionary<ICommand, CommandCallbacks>
        commandToCallbacksMap = new Dictionary<ICommand, CommandCallbacks>();
    #endregion // Data
}

```

```

#region Command Registration
public void RegisterCommand(
    ICommand command, Predicate<object> canExecute, Action<object> execute)
{
    VerifyArgument(command, "command");
    VerifyArgument(canExecute, "canExecute");
    VerifyArgument(execute, "execute");

    _commandToCallbacksMap[command] =
        new CommandCallbacks(canExecute, execute);
}

public void UnregisterCommand(ICommand command)
{
    VerifyArgument(command, "command");

    if (_commandToCallbacksMap.ContainsKey(command))
        _commandToCallbacksMap.Remove(command);
}
#endregion // Command Registration

```

```

#region ICommandSink Members
public virtual bool CanExecuteCommand(
    ICommand command, object parameter, out bool handled)
{
    VerifyArgument(command, "command");
    if (_commandToCallbacksMap.ContainsKey(command))
    {
        handled = true;
        return _commandToCallbacksMap[command].CanExecute(parameter);
    }
    else
    {
        return (handled = false);
    }
}

public virtual void ExecuteCommand(
    ICommand command, object parameter, out bool handled)
{
    VerifyArgument(command, "command");
    if (_commandToCallbacksMap.ContainsKey(command))
    {
        handled = true;
        _commandToCallbacksMap[command].Execute(parameter);
    }
    else
    {
        handled = false;
    }
}
#endregion // ICommandSink Members

```

```

#region VerifyArgument
static void VerifyArgument(object arg, string argName)
{
    if (arg == null)
        throw new ArgumentNullException(argName);
}
#endregion // VerifyArgument

```

```

#region CommandCallbacks [nested struct]
private struct CommandCallbacks
{
    public readonly Predicate<object> CanExecute;
    public readonly Action<object> Execute;

    public CommandCallbacks(Predicate<object> canExecute,
        Action<object> execute)
    {
        this.CanExecute = canExecute;
        this.Execute = execute;
    }
}

```



```

    }
}
#endregion // CommandCallbacks [nested struct]
}

```

## Using CommandSink in ViewModel Classes

Finally, it is time to see how to use **CommandSink** in a ViewModel class. Let's see how the **CommunityViewModel** class is defined. Remember, this class is the ViewModel object assigned to the main **Window**'s **DataContext** property.

```

/// <summary>
/// A ViewModel class that exposes a collection of
/// PersonViewModel objects, and provides a routed
/// command that, when executed, kills the people.
/// This class derives from CommandSink, which is
/// why it does not directly implement the ICommandSink
/// interface. See PersonViewModel for an example
/// of implementing ICommandSink directly.
/// </summary>
public class CommunityViewModel : CommandSink
{
    public CommunityViewModel()
    {
        // Populate the community with some people.
        Person[] people = Person.GetPeople();

        IEnumerable<PersonViewModel> peopleView =
            people.Select(p => new PersonViewModel(p));

        this.People = new ReadOnlyCollection<PersonViewModel>(peopleView.ToArray());

        // Register the command that kills all the people.
        base.RegisterCommand(
            KillAllMembersCommand,
            param => this.CanKillAllMembers,
            param => this.KillAllMembers());
    }

    public ReadOnlyCollection<PersonViewModel> People { get; private set; }

    public static readonly RoutedCommand KillAllMembersCommand = new RoutedCommand();

    public bool CanKillAllMembers
    {
        get { return this.People.Any(p => p.CanDie); }
    }

    public void KillAllMembers()
    {
        foreach (PersonViewModel personView in this.People)
            if (personView.CanDie)
                personView.Die();
    }
}

```

The magic occurs in the constructor. Notice the call to the **RegisterCommand** method, which is defined by the **CommandSink** base class. The first argument is the command being registered. The second argument is a **Predicate<object>**, created as a lambda expression, which is invoked when the command is queried to see if it can execute. The last argument is an **Action<object>**, created as a lambda expression, which is invoked when the command executes. Naturally, the use of lambda expressions is optional.

Now, let's see how the **PersonViewModel** class works. It is an example of embedding **CommandSink**, instead of deriving from it. Since **PersonViewModel** does not derive from **CommandSink**, it must implement the **ICommandSink** interface instead.

Below is the constructor of **PersonViewModel**:

```

readonly CommandSink _commandSink;

```

```

public PersonViewModel(Person person)
{
    _person = person;

    _commandSink = new CommandSink();

    _commandSink.RegisterCommand(
        DieCommand,
        param => this.CanDie,
        param => this.Die());

    _commandSink.RegisterCommand(
        SpeakCommand,
        param => this.CanSpeak,
        param => this.Speak(param as string));
}

```

I won't bother showing you every member of the class, since they are not very important for this review. We will, however, now see how **PersonViewModel** implements the  **ICommandSink** interface and how it uses an instance of the **CommandSink** class.

```

public bool CanExecuteCommand(ICommand command, object parameter, out bool handled)
{
    return _commandSink.CanExecuteCommand(command, parameter, out handled);
}

public void ExecuteCommand(ICommand command, object parameter, out bool handled)
{
    _commandSink.ExecuteCommand(command, parameter, out handled);
}

```

## Revision History

- July 26, 2008 – Made several improvements to the solution, including: took [Bill Kempf](#)'s advice and created the **CommandSink** class, updated the **ViewModel** classes to use it, renamed **RelayCommandBinding** to **CommandSinkBinding**, and fixed **CommandSinkBinding** so that it works for elements in a template. Updated the article and source code download.
- July 24, 2008 – Created the article.

## License



This article, along with any associated source code and files, is licensed under [The Code Project Open License \(CPOL\)](#)

## Share

## About the Author



**Josh Smith**


 Software Developer (Senior) Cynergy Systems  
 United States 

Josh creates software, for iOS and Windows.

He works at [Cynergy Systems](#) as a Senior Experience Developer.

Read his [iOS Programming for .NET Developers](#) book to learn how to write iPhone and iPad apps by leveraging your existing .NET skills.

Use his [Master WPF](#) app on your iPhone to sharpen your WPF skills on the go.

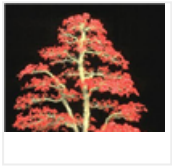
Check out his [Advanced MVVM](#) book.

Visit his [WPF blog](#) or stop by his [iOS blog](#).

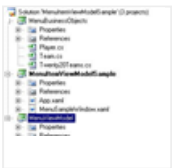
## You may also be interested in...



[Sinking RoutedCommands to ViewModel Commands in WPF](#)



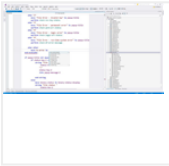
[Simplifying the WPF TreeView by Using the ViewModel Pattern](#)



[WPF Menu using ViewModel - Part 1](#)



[Microsoft's Guide to Modern Dev/Test](#)




[10 Ways to Boost COBOL Application Development](#)



[Getting to Know the Arduino 101 Platform](#)

## Comments and Discussions

 **132 messages** have been posted for this article Visit <http://www.codeproject.com/Articles/28093/Using-RoutedCommands-with-a-ViewModel-in-WPF> to post and view comments on this article, or click [here](#) to get a print view with messages.

[Permalink](#) | [Advertise](#) | [Privacy](#) | [Terms of Use](#) | Mobile  
Web02 | 2.8.160621.1 | Last Updated 24 Jul 2008

Select Language ▼

Article Copyright 2008 by Josh Smith  
Everything else Copyright © [CodeProject](#), 1999-2016