Understanding Routed Commands

This blog post explains what routed commands are, and why you should use them. My explanation does not cover all of the details, nor does it show all of the various uses of routed commands. For a more comprehensive review of commanding, but with a very different (and somewhat misleading) perspective on the issue, be sure to check out the official documentation here (http://msdn2.microsoft.com/en-us/library/ms752308.aspx) (http://msdn2.microsoft.com/en-us/library/ms752308.aspx).

Many demonstrations of routed commands explain them in the context of text editing, which is just one of the many ways to use them. In fact, so many explanations about routed commands focus on how to use them in text editing scenarios that many people I have met assume routed commands have no practical use outside of text editing. That is an incorrect, yet understandable, interpretation of one of WPF's coolest features.

What is a routed command? Well, the answer to that question is too complicated to explain without first taking a step back and explaining some other things. To understand what a routed command is, we will examine what "routed" means, and then what "command" means. Once we have those two ideas defined, we can combine them and see what happens.

Routed

In WPF, the visual elements that constitute a user interface form a hierarchical tree structure. This "element tree" has many purposes, and is a core part of how WPF operates. The root node in the tree is a container, such as a Window or a Page. Everything seen within that top-level container is a descendant element in its element tree. There are two "views" of this element tree: the visual tree and logical tree. To learn more about that concept, I recommend you read this article (http://www.codeproject.com/KB/WPF/WpfElementTrees.aspx).

In general, the term "route" refers to a path between two points in a network. Take that idea and apply it to the WPF element tree, and now you have a path between visual elements in a user interface. Now imagine an electric current quickly flowing through that route, zapping each element it passes through. When an element is zapped by the electric current, it has the opportunity to come to life and do whatever it wants (i.e. execute an event handler). That is pretty much how I think about routed events, except my complete mental model involves unicorns, robots, and laser beams, as seen in the diagram below:

…just kidding.

One could provide a more mundane definition of a routed event as a notification that travels a route between two elements in the WPF visual tree. They are not *really* events, in the standard sense of .NET events. People wrap routed events with standard .NET events, out of convention, just to make them more usable and ".NETish." But in reality, routed events are not events. They are a feature of the WPF framework, intimately tied to the WPF visual tree.

Routed commands use routed events, as do many other parts of WPF, so it is important to understand them thoroughly. If you are still hazy on the idea of routed events, check out the blog post I wrote about the topic here (https://joshsmithonwpf.wordpress.com/2007/06/22/overview-of-routed-events-in-wpf/).

Command

The Command pattern is nothing new. It means that you create an object that knows how to do some task, and then the entire application relies on that object to do that task. For example, if I am creating an image editing application, I could create a command object called "CreateNewImage" or something like that. When the user either clicks the "New" menu item, or clicks the "New" toolbar button, or hits Ctrl + N, etc., the same CreateNewImage command object would create a new image.

In WPF, we have the ICommand (http://msdn2.microsoft.com/en-us/library/system.windows.input.icommand.aspx) interface, which makes it easy to create command objects that WPF classes can use. When you implement ICommand, you place all the knowledge of what needs to be done to accomplish the task in the command. You can then assign that command to, say, a Button's Command property so that the command will execute when the user clicks the Button.

The ICommand interface has three members that all commands must define. It has two methods and one event. The CanExecute method determines if the state of the application currently supports the execution of the command. If CanExecute returns true, then the Execute method can be invoked, which is what actually performs the execution logic baked into the command. If CanExecute returns false, all controls whose Command references that command object are disabled so that the user cannot attempt to execute the command at that time. This is a very convenient feature, especially if multiple controls in the user interface all are bound to that command.

If the command determines that it's "can execute" status changes, it can raise the CanExecuteChanged event, defined by the ICommand interface. This lets WPF know that it should call the command's CanExecute method again, to query the new status.

Routed + Command = RoutedCommand

So what happens when you mix routed events with commands? Well, you get a very powerful way to think about how an application represents and implements its features. You get routed commands.

A routed command is a command object that does not know how to accomplish the task it represents. It simply represents the task/feature. When asked if it can execute and when told to execute, it simply delegates that responsibility off to somewhere else.

The answer is, confusingly enough, it does not know whom.

A routed command does not determine if it can execute and what to do when executed. Instead, some routed events travel through the element tree, giving any element in the UI a chance to do the command's work for it. It truly is just a semantic identifier: a named entity that represents a feature of a program.

When a routed command is asked if it can execute, the routed PreviewCanExecute and CanExecute events tunnel down and bubble up the element tree. These events give all elements between the root of the tree (such as a Window) and the source element that references the command a chance to determine if the command can execute. When the command is told to execute, the PreviewExecuted and Executed routed events travel the same event route, checking to see if anybody cares to react to the event. If so, they can run a handler for the event, if not, the event finishes zapping the elements and nothing happens.

Who Cares?

You might be wondering why it is a good idea to use routed commands at all. Why bother? What's wrong with just hooking a Button's Click event and doing things the "normal" way?

Well, you do not have to use routed commands if you do not want to. You certainly can just hook a Button's Click event and go to town. By extension, why bother with a data layer and a business layer? Why not just stick your whole application into Window1.xaml.cs and be done with it? That would be much easier, right?

My point is, there are very real advantages to adhering to the loose coupling proffered by routed commands. Let's review them.

First, all controls using the same RoutedCommand will automatically be disabled when the command cannot execute. If you have ever used a program where clicking on enabled buttons does not actually do anything, you will appreciate this.

Second, there is less event handler code to write since most of the wiring is provided for you by the commanding system. You do not have to add event handlers for each UI element that executes the same command, whereas using events directly off of UI elements requires many handlers that all basically do the same thing.

Third, if you use my implementation of Model-View-Controller or Structural Skinning, using routed commands is an absolute must. Learn more about MVC here (http://www.codeproject.com/KB/WPF/MVCtoUnitTestinWPF.aspx) and Structural Skinning here

(http://www.codeproject.com/KB/WPF/podder2.aspx).

Fourth, using routed commands makes it possible to decouple the Software Engineering team from the Visual Design team. The developers don't have to worry about what type of element is consuming application functionality, just as long as the UI executes the right commands all is well. This also frees the designers from having to worry about such details so that they can focus on creating a great user experience.

Fifth, but certainly not last, using routed commands as part of your design process forces you to map functional requirements to commands up front. This process of taking a list of required features and translating them into RoutedCommand objects with meaningful names is invaluable to help the team as a whole understand the system they are about to build. It also creates a set of short terms that people can use to refer to features.

Show me Some Code

I threw together a quick demo project in Visual Studio 2008, showing the simplest possible usage of a custom routed command. Download the code here: Routed Command Demo (https://joshsmithonwpf.files.wordpress.com/2008/03/routedcommanddemozip.doc) Be sure to rename the file extension from .DOC to .ZIP and then decompress the file.

This entry was posted on Tuesday, March 18th, 2008 at 7:12 am and is filed under Theoria. You can follow any responses to this entry through the RSS 2.0 feed. Both comments and pings are currently closed.

29 Responses to *Understanding Routed Commands*

Neil Mosafi says:
March 18, 2008 at 9:43 am
Hey nice article Josh and very well illustrated!

Another thing about Routed commands which I love is that they all go through the CommandManager to do their work. One advantage of this is that command manager will automatically cause all active routed command's CanExecute properties to be reevaluated.

Contrast this to manually implementing the ICommand interface where you have raise that event yourself (which could be due to a number of unrelated things changing). I simplied my ViewModel code a lot (removed loads of spaghetti) when I changed from using raw ICommand implementations to binding to RoutedCommands.

Josh Smith says:
March 18, 2008 at 10:11 am
That is a great point, Neil. Thanks for raising it. I actually intended to mention how you can programmatically cause the commands to be requeried for "can execute" status, but forgot to. So, for those of you reading this comment, if you find yourself wishing that a certain routed command would have it's CanExecute method checked by WPF, call CommandManager.InvalidateRequerySuggested().

marlongrech says:

March 18, 2008 at 10:23 am
great post dude…. your posts are always getting better

Michael Brown says:
March 18, 2008 at 11:45 am
Josh,
Excellent overview of Routed Commands. The illustrations definitely made me chuckle.

Jonah Simpson says:
March 18, 2008 at 11:46 am
the unicorn reference….hahaha

Josh Smith says:
March 18, 2008 at 11:46 am
Thanks Marlon, Mike, and Jonah.

You guys rock! Long live the WPF Disciples.

Karl Shifflett says:
March 18, 2008 at 3:18 pm
Josh,

Thank you for this very creative article on RoutedCommands. I'm glad you keep preaching RoutedCommands because they are the BOMB!

Best to you,

Karl

Josh Smith says:
March 18, 2008 at 3:23 pm
You know it, buddy. I am just trying to form an army of Commanders, so that we can take over the world. It's just a matter of time…

Corrado Cavalli says:
March 18, 2008 at 4:35 pm
Great article Josh!

Josh Smith says:
March 18, 2008 at 7:49 pm
Thanks a lot, Corrado. I had fun writing this one.

chong says:
March 19, 2008 at 8:22 am
One limitation of routedcommand is that it is only routed to visual element.
Many times, it is better to let controller (instead of view) to handle the command.

This is similar idea to event hub in CAB.

Josh Smith says:
March 19, 2008 at 8:27 am
chong,

The view can easily delegate the call to the controller, or the CommandBinding for that command can use controller methods to handle the CanExecute and Executed events.

Thanks,
Josh

Kent Boogaart says:
March 20, 2008 at 6:15 am
Great post Josh. One thing though: your 'Who Cares?' section suggests a choice between using routed commands and handling click events. I think the harder choice is 'do I use routed commands or standard commands?'. All of your arguments for using routed commands apply equally well for standard ICommand implementations. Even the designer/developer collaboration issue can be solved using standard ICommands, as long as those commands are agreed upon up front and the right UI components are wired to the right ICommand instance.

To me, routed commands really shine when developing custom controls because you don't know who will be consuming your control, nor what they want to do in response to user actions. So your control can just execute a routed command and let the control consumer deal with handling that command.

Anyways, just a thought. Keep up the great blogging!

sacha says:
March 20, 2008 at 6:39 am
Excellent as always…where do you get these images…

Josh Smith says:
March 20, 2008 at 8:12 am
Kent,

I see what you mean, but must disagree. There is a big "gotcha" around implementing ICommand that can sneak up and bite you in a big way. If you are using the ICommand approach, instead of routed commands, then you have to use the WeakEvent pattern in your CanExecuteChanged event. This is because visual elements will hook that event, and since the command object might never be garbage collected until the app shuts down, there is a very real potential for a memory leak. Since the elements do not automatically unhook the CanExecuteChanged event on your command objects, they will continue to be referenced (and, hence, not eligible for garbage collection) until the app shuts down. This can be a nasty memory leak if your app creates many transient UI elements.

Thanks,
Josh

Josh Smith says:
March 20, 2008 at 8:13 am
Thanks Sacha! I get the images from Google Images, and then add some text callouts using SnagIt.

Kent Boogaart says:
March 20, 2008 at 10:45 am
Josh,

That can be solved simply by using an abstract base class for all your commands that implements the weak event pattern. Anyways, there are advantages to both approaches and I'm not suggesting to always use one or the other – just to choose the right one for the task at hand.

Thanks again,
Kent

PS. Any chance you're working on the Infra WPF grid?

David Avera says:
April 1, 2008 at 3:02 pm
Josh,

I downloaded and ran the sample project mentioned in this article. When it ran, every single window available in VS opened … from data sources to pending checkins. I don't think I did anything other than unzip and run it in the debugger. Now, every instance of VS does the same thing. I have tried resetting layouts, closing the windows and exiting VS … but nothing seems to change the behavior. I tried asking some coworkers and then tried resetting my setting … but it still does it. Could you possibly help with this? I would greatly appreciate it. If you wish you may respond to my email addres. Thanks very much.

Josh Smith says:
April 1, 2008 at 3:03 pm
David,

Sorry, I have no idea what the problem is. That's very strange.

Josh

David Avera says:
April 1, 2008 at 4:23 pm
Josh,
Thanks for the reply. The problem appears to be corrected with a system restart. I >think< something got hosed and there was a devenv.exe process that wasn't going away when I shut down all instances of VS. During shutdown I had to kill it. So, it appears better now.

Mike Demopoulos says:
April 1, 2008 at 7:06 pm
Awesome post. I'm a Flash developer who is trying to make the transition over to WPF and it's articles like these that really help. …i have to admit, my head is still spinning, but it's getting easier. Thanks!

WPF Dev - Josh Smith, post happy says:
April 1, 2008 at 10:55 pm
[…] Images in a 3D ItemsControl – Creating a custom 3D panel Understanding Routed Commands – This one blew my mind a little bit. Maybe it was the unicorns, robots, and laser beams. […]

Andres Olivares says:
April 29, 2008 at 9:33 pm
Josh,

Great article, and I agree the images are funny. I was interested in a scenario Kent mentioned "commands really shine when developing custom controls." I was looking for a way to make commands for a component executed somewhere centralized away from the component, even away from the client implementing the component.

Scenario:
I have a Window [Win1] and it contains a component [Comp1]. Comp1 contains a menu item and it is bound to a central command. When executed Win1 determines whether it can be executed but the command executes in

another class, like a service, and it works of some context. This would be ideal because I can then code this CommandController similar to a service for a specific context. This imaginery CommandController would have exposed commands and handlers for those commands. The final goal in this scenario would be that another component, possibly in another Window, could bind to the same command because it would want to deal with the same context identically.

Hope this make sense. I am still in search to make this happen and I believe your article helped me understand the usage of routed commands and my limitations.

Thanks again.

Patrick Duffy says:
May 7, 2008 at 11:36 am
Hello Josh,

I was reading your article on WPF commanding and I have a question. I have an application that has a complex layout with multiple tab windows imbedded in a document container window which hosts a flowdocument. When I create a command for 'cut', 'copy' and 'paste' (from the ApplicationCommands class) and place it in a tool bar menu, the 'tunnel and bubble' fails to enable these commands when I select text in the flowdocument, but if I place the menu directly iside the xaml element that holds the flow document it works. Why does WPF fail to bubble up to the top-level toolbar but it works if I place the menu farther down the tree? Any idea on what to do to fix it?

Josh Smith says:
May 7, 2008 at 12:49 pm
I dunno. You should search the WPF Forum, and if you can't find an answer there, post your question on it.
http://forums.microsoft.com/MSDN/ShowForum.aspx?ForumID=119&SiteID=1

Josh

Yogesh says:
July 12, 2008 at 9:07 am
Hello Josh,

Can you help me with Application wide commands?

Does any such term exists? What I want is, that I define CanExecute and Execute handlers on a application/top window/main frame level and all the child windows or pages of main frame pick those handlers automatically. Is this possible in any way?

If you got confused with what I want, I want something similar to what is being discussed here:
http://forums.msdn.microsoft.com/en/wpf/thread/cc2645a3-014b-4244-a91c-bbe40872cee0/

Regards,
Yogesh.

Josh Smith says:
July 12, 2008 at 9:18 am
Yogesh,

I suggest using CommandManager.RegisterClassCommandBinding().

Josh

Dog Tate says:
August 27, 2008 at 8:35 pm
That unicorn riding robot is now living on my desktop. Thanks!

And oh yeah, thanks for the great post too. I've been banging my head against a bunch of MVVM WPF examples and trying to understand what the hell the difference was between the implementations. Recognizing the difference between ICommand and RoutedCommand implementations was the key! Woo freakin hoo. Great post.

Josh Smith says:
August 27, 2008 at 8:56 pm
I'm glad to be of service, Dog! 8)