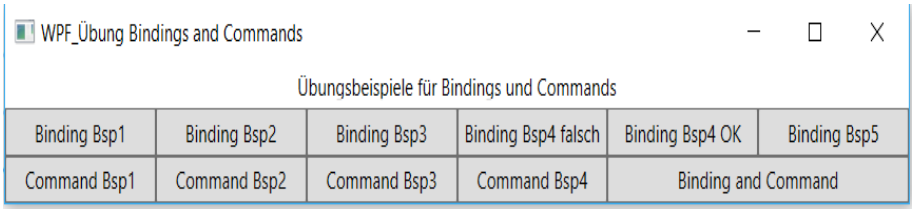
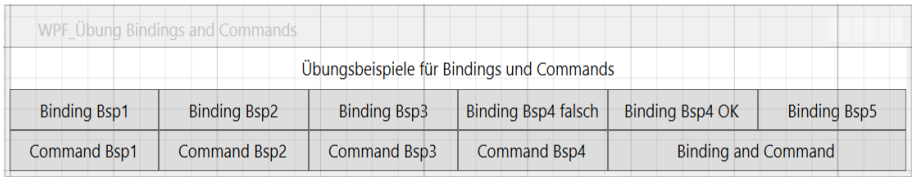
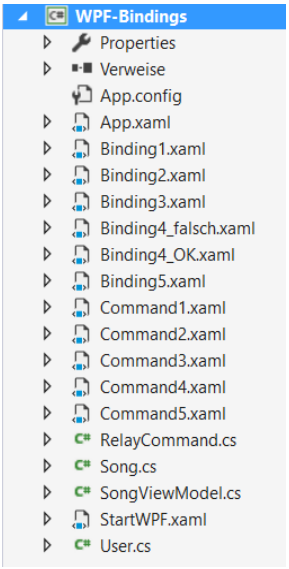
WPF Übung Bindungen

Erstelle ein neues WPF-Projekt „BindingsAndCommands“. Erzeuge das folgende StartFenster „StartWPF“. Von diesem Fenster aus können dann die verschiedenen Beispiele (Bindings Bsp1-5) per Mausklick aufgerufen werden. Für die Klasse User ist eine eigene Datei zu erstellen. Die Namespaces bzw. Klassennamen der folgenden Listings müssen eventuell angepasst werden.

Startfenster StartWPF



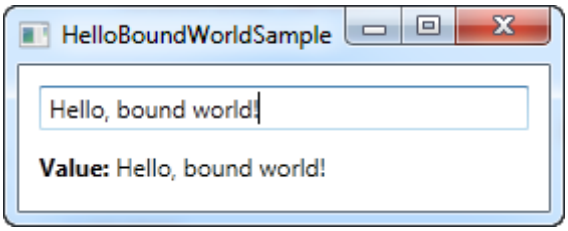




WPF Binding – Bsp1

Das folgende Beispiel soll mit dem Button „Binding Bsp1“ gestartet werden.

```
<Window x:Class="WpfTutorialSamples.DataBinding.HelloBoundWorldSample"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="HelloBoundWorldSample" Height="110" Width="280">
    <StackPanel Margin="10">
                <TextBox Name="txtValue" />
                <WrapPanel Margin="0,10">
                        <TextBlock Text="Value: " FontWeight="Bold" />
                        <TextBlock Text="{Binding Path=Text, ElementName=txtValue}" />
                </WrapPanel>
        </StackPanel>
</Window>
```



This simple example shows how we bind the value of the TextBlock to match the Text property of the TextBox. As you can see from the screenshot, the TextBlock is automatically updated when you enter text into the TextBox. In a non-bound world, this would require us to listen to an event on the TextBox and then update the TextBlock each time the text changes, but with data binding, this connection can be established just by using markup.

The syntax of a Binding

All the magic happens between the curly braces, which in XAML encapsulates a Markup Extension. For data binding, we use the Binding extension, which allows us to describe the binding relationship for the Text property. In its most simple form, a binding can look like this:

{Binding}

This simply returns the current data context (more about that later). This can definitely be useful, but in the most common situations, you would want to bind a property to another property on the data context. A binding like that would look like this:

{Binding Path=NameOfProperty}

The Path notes the property that you want to bind to, however, since Path is the default property of a binding, you may leave it out if you want to, like this:

{Binding NameOfProperty}

You will see many different examples, some of them where Path is explicitly defined and some where it's left out. In the end it's really up to you though.

A binding has many other properties though, one of them being the ElementName which we use in our example. This allows us to connect directly to another UI element as the source. Each property that we set in the binding is separated by a comma:

{Binding Path=Text, ElementName=txtValue}

WPF Binding – Bsp2

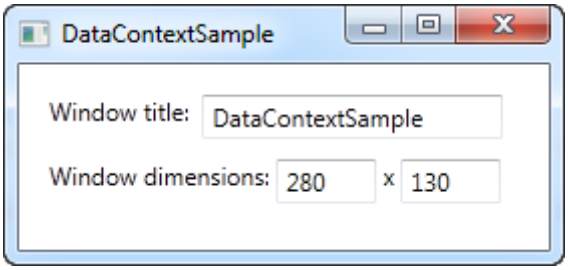
Das folgende Beispiel soll mit dem Button „Binding Bsp2“ gestartet werden.

The DataContext property is the default source of your bindings, unless you specifically declare another source, like we did in the previous chapter with the ElementName property. It's defined on the FrameworkElement class, which most UI controls, including the WPF Window, inherits from. Simply put, it allows you to specify a basis for your bindings

There's no default source for the **DataContext** property (it's simply null from the start), but since a DataContext is inherited down through the control hierarchy, you can set a DataContext for the Window itself and then use it throughout all of the child controls. Let's try illustrating that with a simple example:

```
<Window x:Class="WpfTutorialSamples.DataBinding.DataContextSample"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="DataContextSample" Height="130" Width="280">
        <StackPanel Margin="15">
                <WrapPanel>
                        <TextBlock Text="Window title:  " />
                        <TextBox Text="{Binding Title, UpdateSourceTrigger=PropertyChanged}"
Width="150" />
                </WrapPanel>
                <WrapPanel Margin="0,10,0,0">
                        <TextBlock Text="Window dimensions: " />
                        <TextBox Text="{Binding Width}" Width="50" />
                        <TextBlock Text=" x " />
                        <TextBox Text="{Binding Height}" Width="50" />
                </WrapPanel>
        </StackPanel>
</Window>
```

```
using System;
using System.Windows;

namespace WpfTutorialSamples.DataBinding
{
        public partial class DataContextSample : Window
        {
                public DataContextSample()
                {
                        InitializeComponent();
                        this.DataContext = this;
                }
        }
}
```



The Code-behind for this example only adds one line of interesting code: After the standard InitalizeComponent() call, **we assign the "this" reference to the DataContext**, which basically just tells the Window that we want itself to be the data context.

In the XAML, we use this fact to bind to several of the Window properties, including Title, Width and Height. Since the window has a DataContext, which is passed down to the child controls, we don't have to define a source on each of the bindings - we just use the values as if they were globally available.

Try running the example and resize the window - you will see that the dimension changes are immediately reflected in the textboxes. You can also try writing a different title in the first textbox, but you might be surprised to see that this change is not reflected immediately. Instead, you have to move the focus to another control before the change is applied. Why? Well, that's the subject for the next chapter.

Summary

Using the DataContext property is like setting the basis of all bindings down through the hierarchy of controls. This saves you the hassle of manually defining a source for each binding, and once you really start using data bindings, you will definitely appreciate the time and typing saved.

However, this doesn't mean that you have to use the same DataContext for all controls within a Window. Since each control has its own DataContext property, you can easily break the chain of inheritance and override the DataContext with a new value. This allows you to do stuff like having a global DataContext on the window and then a more local and specific DataContext on e.g. a panel holding a separate form or something along those lines.

The UpdateSourceTrigger property – Bsp3

Das folgende Beispiel soll mit dem Button „Binding Bsp3“ gestartet werden.

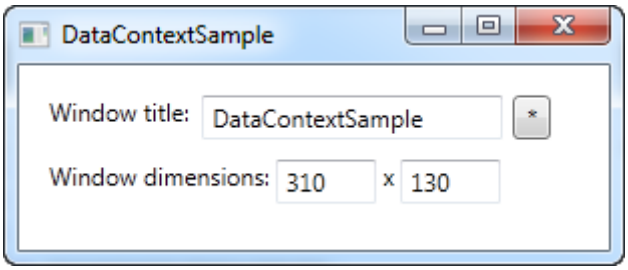
In the previous article we saw how changes in a TextBox was not immediately sent back to the source. Instead, the source was updated only after focus was lost on the TextBox. This behavior is controlled by a property on the binding called **UpdateSourceTrigger**. It defaults to the value "**Default**", which basically means that the source is updated based on the property that you bind to. As of writing, all properties except for the Text property, is **updated as soon as the property changes (PropertyChanged)**, while the Text property is updated when focus on the destination element is lost (LostFocus).
Default is, obviously, the default value of the UpdateSourceTrigger. The other options are **PropertyChanged**, **LostFocus** and **Explicit**. The first two has already been described, while the last

one simply means that the update has to be pushed manually through to occur, using a call to UpdateSource on the Binding.

To see how all of these options work, I have updated the example from the previous chapter to show you all of them:

```
<Window x:Class="WpfTutorialSamples.DataBinding.DataContextSample"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="DataContextSample" Height="130" Width="310">
        <StackPanel Margin="15">
                <WrapPanel>
                        <TextBlock Text="Window title:  " />
                        <TextBox Name="txtWindowTitle" Text="{Binding Title,
UpdateSourceTrigger=Explicit}" Width="150" />
                        <Button Name="btnUpdateSource" Click="btnUpdateSource Click"
Margin="5,0" Padding="5,0">*</Button>
                </WrapPanel>
                <WrapPanel Margin="0,10,0,0">
                        <TextBlock Text="Window dimensions: " />
                        <TextBox Text="{Binding Width, UpdateSourceTrigger=LostFocus}"
Width="50" />
                        <TextBlock Text=" x " />
                        <TextBox Text="{Binding Height,
UpdateSourceTrigger=PropertyChanged}" Width="50" />
                </WrapPanel>
        </StackPanel>
</Window>
```

```
using System;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Data;

namespace WpfTutorialSamples.DataBinding
{
        public partial class DataContextSample : Window
        {
                public DataContextSample()
                {
                        InitializeComponent();
                        this.DataContext = this;
                }

                private void btnUpdateSource_Click(object sender, RoutedEventArgs e)
                {
                        BindingExpression binding =
txtWindowTitle.GetBindingExpression(TextBox.TextProperty);
                        binding.UpdateSource();
                }
        }
}
```



As you can see, each of the three textboxes now uses a different **UpdateSourceTrigger**.
The first one is set to **Explicit**, which basically means that the source won't be updated unless you manually do it. For that reason, I have added a button next to the TextBox, which will update the source value on demand. In the Code-behind, you will find the Click handler, where we use a couple of lines of code to get the binding from the destination control and then call the UpdateSource() method on it.
The second TextBox uses the **LostFocus** value, which is actually the default for a Text binding. It means that the source value will be updated each time the destination control loses focus.

The third and last TextBox uses the **PropertyChanged** value, which means that the source value will be updated each time the bound property changes, which it does in this case as soon as the text changes.

Try running the example on your own machine and see how the three textboxes act completely different: The first value doesn't update before you click the button, the second value isn't updated until you leave the TextBox, while the third value updates automatically on each keystroke, text change etc.

Summary

The UpdateSourceTrigger property of a binding controls how and when a changed value is sent back to the source. However, since WPF is pretty good at controlling this for you, the default value should suffice for most cases, where you will get the best mix of a constantly updated UI and good performance.

For those situations where you need more control of the process, this property will definitely help though. Just make sure that you don't update the source value more often than you actually need to. If you want the full control, you can use the **Explicit** value and then do the updates manually, but this does take a bit of the fun out of working with data bindings.

Responding to changes – Bsp4_falsch

Das folgende Beispiel soll mit dem Button „Binding Bsp4_falsch“ gestartet werden.

So far in this tutorial, we have mostly created bindings between UI elements and existing classes, but in real life applications, you will obviously be binding to your own data objects. This is just as easy, but once you start doing it, you might discover something that disappoints you: Changes are not automatically reflected, like they were in previous examples. As you will learn in this article, you need just a bit of extra work for this to happen, but fortunately, WPF makes this pretty easy.

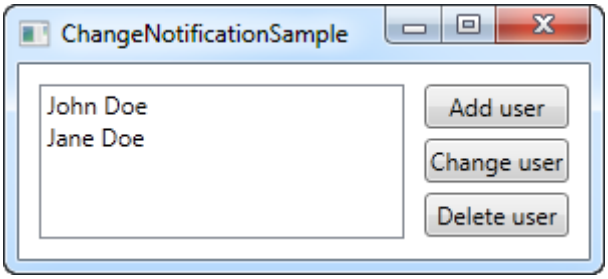
Responding to data source changes

There are two different scenarios that you may or may not want to handle when dealing with data source changes: Changes to the list of items and changes in the bound properties in each of the data objects. How to handle them may vary, depending on what you're doing and what you're looking to accomplish, but WPF comes with two very easy solutions that you can use: The **ObservableCollection** and the **INotifyPropertyChanged** interface. **The following example will show you why we need these two things:**

```
<Window x:Class="WpfTutorialSamples.DataBinding.ChangeNotificationSample"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="ChangeNotificationSample" Height="150" Width="300">
        <DockPanel Margin="10">
                <StackPanel DockPanel.Dock="Right" Margin="10,0,0,0">
                        <Button Name="btnAddUser" Click="btnAddUser Click">Add user</Button>
                        <Button Name="btnChangeUser" Click="btnChangeUser Click"
Margin="0,5">Change user</Button>
                        <Button Name="btnDeleteUser" Click="btnDeleteUser Click">Delete
user</Button>
                </StackPanel>
                <ListBox Name="lbUsers" DisplayMemberPath="Name"></ListBox>
        </DockPanel>
</Window>
```

```
using System;
using System.Collections.Generic;
using System.Windows;

namespace WpfTutorialSamples.DataBinding
{
        public partial class ChangeNotificationSample : Window
        {
                private List<User> users = new List<User>();

                public ChangeNotificationSample()
                {
```

```
                        InitializeComponent();

                        users.Add(new User() { Name = "John Doe" });
                        users.Add(new User() { Name = "Jane Doe" });

                        lbUsers.ItemsSource = users;
                }

                private void btnAddUser Click(object sender, RoutedEventArgs e)
                {
                        users.Add(new User() { Name = "New user" });
                }

                private void btnChangeUser Click(object sender, RoutedEventArgs e)
                {
                        if(lbUsers.SelectedItem != null)
                                (lbUsers.SelectedItem as User).Name = "Random Name";
                }

                private void btnDeleteUser Click(object sender, RoutedEventArgs e)
                {
                        if(lbUsers.SelectedItem != null)
                                users.Remove(lbUsers.SelectedItem as User);
                }
        }

        public class User
        {
                public string Name { get; set; }
        }
}
```



Try running it for yourself and watch how even though you add something to the list or change the name of one of the users, **nothing in the UI is updated**. The example is pretty simple, with a User class that will keep the name of the user, a ListBox to show them in and some buttons to manipulate both the list and its contents. The ItemsSource of the list is assigned to a quick list of a couple of users that we create in the window constructor. The problem is that none of the buttons seems to work. Let's fix that, in two easy steps.

Responding to changes – Bsp4_OK

Das folgende Beispiel soll mit dem Button „Binding Bsp4_OK“ gestartet werden.

The first step is to get the UI to respond to changes in the list source (ItemsSource), like when we add or delete a user. What we need is a list that notifies any destinations of changes to its content, and fortunately, WPF provides a type of list that will do just that. It's called ObservableCollection, and you use it much like a regular List<T>, with only a few differences.

In the final example, which you will find below, we have simply replaced the List<User> with an ObservableCollection<User> - that's all it takes! This will make the Add and Delete button work, but it won't do anything for the "Change name" button, because the change will happen on the bound data object itself and not the source list - the second step will handle that scenario though.

Reflecting changes in the data objects

The second step is to let our custom User class implement the INotifyPropertyChanged interface. By doing that, our User objects are capable of alerting the UI layer of changes to its properties. This is a bit more cumbersome than just changing the list type, like we did above, but it's still one of the simplest way to accomplish these automatic updates.

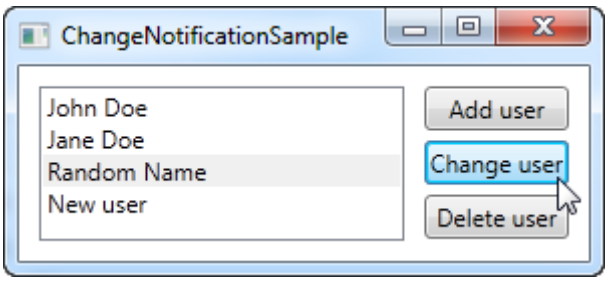
The final and working example

With the two changes described above, we now have an example that WILL reflect changes in the data source. It looks like this:

```
<Window x:Class="WpfTutorialSamples.DataBinding.ChangeNotificationSample"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="ChangeNotificationSample" Height="135" Width="300">
        <DockPanel Margin="10">
                <StackPanel DockPanel.Dock="Right" Margin="10,0,0,0">
                        <Button Name="btnAddUser" Click="btnAddUser_Click">Add user</Button>
                        <Button Name="btnChangeUser" Click="btnChangeUser Click"
Margin="0,5">Change user</Button>
                        <Button Name="btnDeleteUser" Click="btnDeleteUser Click">Delete
user</Button>
                </StackPanel>
                <ListBox Name="lbUsers" DisplayMemberPath="Name"></ListBox>
        </DockPanel>
</Window>
```

```
using System;
using System.Collections.Generic;
using System.Windows;
using System.ComponentModel;
using System.Collections.ObjectModel;

namespace WpfTutorialSamples.DataBinding
{
        public partial class ChangeNotificationSample : Window
        {
                private ObservableCollection<User> users = new ObservableCollection<User>();

                public ChangeNotificationSample()
                {
                        InitializeComponent();

                        users.Add(new User() { Name = "John Doe" });
                        users.Add(new User() { Name = "Jane Doe" });

                        lbUsers.ItemsSource = users;
                }

                private void btnAddUser Click(object sender, RoutedEventArgs e)
                {
                        users.Add(new User() { Name = "New user" });
                }

                private void btnChangeUser Click(object sender, RoutedEventArgs e)
                {
                        if(lbUsers.SelectedItem != null)
                                (lbUsers.SelectedItem as User).Name = "Random Name";
                }

                private void btnDeleteUser Click(object sender, RoutedEventArgs e)
                {
                        if(lbUsers.SelectedItem != null)
                                users.Remove(lbUsers.SelectedItem as User);
                }
        }

        public class User : INotifyPropertyChanged
        {
                private string name;
                public string Name {
```

```
                        get { return this.name; }
                        set
                        {
                                if(this.name != value)
                                {
                                        this.name = value;
                                        this.NotifyPropertyChanged("Name");
                                }
                        }
                }

                public event PropertyChangedEventHandler PropertyChanged;

                public void NotifyPropertyChanged(string propName)
                {
                        if(this.PropertyChanged != null)
                                this.PropertyChanged(this, new
PropertyChangedEventArgs(propName));
                }
        }
}
```



Summary

As you can see, implementing INotifyPropertyChanged is pretty easy, but it does create a bit of extra code on your classes, and adds a bit of extra logic to your properties. This is the price you will have to pay if you want to bind to your own classes and have the changes reflected in the UI immediately. Obviously you only have to call NotifyPropertyChanged in the setter's of the properties that you bind to - the rest can remain the way they are.

The ObservableCollection on the other hand is very easy to deal with - it simply requires you to use this specific list type in those situations where you want changes to the source list reflected in a binding destination.

Value conversion with IValueConverter – Bsp5

Das folgende Beispiel soll mit dem Button „Binding Bsp5“ gestartet werden.

So far we have used some simple data bindings, where the sending and receiving property was always compatible. However, you will soon run into situations where you want to use a bound value of one type and then present it slightly differently.

When to use a value converter

Value converters are very frequently used with data bindings. Here are some basic examples:

- You have a numeric value but you want to show zero values in one way and positive numbers in another way
- You want to check a CheckBox based on a value, but the value is a string like "yes" or "no" instead of a Boolean value
- You have a file size in bytes but you wish to show it as bytes, kilobytes, megabytes or gigabytes based on how big it is

These are some of the simple cases, but there are many more. For instance, you may want to check a checkbox based on a Boolean value, but you want it reversed, so that the CheckBox is checked if the value is false and not checked if the value is true. You can even use a converter to generate an image for an ImageSource, based on the value, like a green sign for true or a red sign for false - the possibilities are pretty much endless!

For cases like this, you can use a value converter. These small classes, which implement the IValueConverter interface, will act like middlemen and translate a value between the source and the destination. So, in any situation where you need to transform a value before it reaches its destination or back to its source again, you likely need a converter.

Implementing a simple value converter

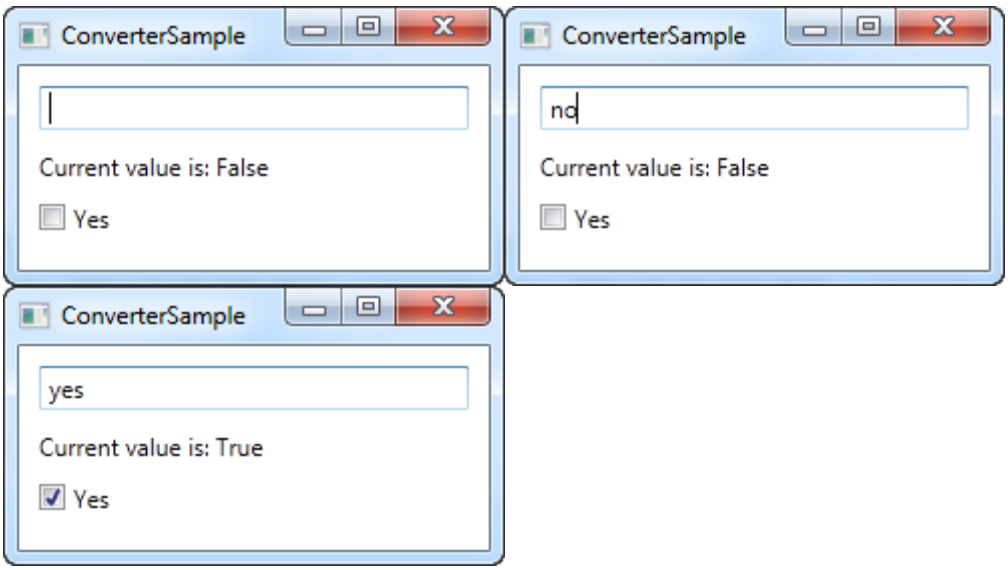
As mentioned, a WPF value converter **needs to implement the IValueConverter interface**, **or alternatively, the IMultiValueConverter interface** (more about that one later). Both interfaces just requires you to implement two methods: Convert() and ConvertBack(). As the name implies, these methods will be used to convert the value to the destination format and then back again.

Let's implement a simple converter which takes a string as input and then returns a Boolean value, as well as the other way around. If you're new to WPF, and you likely are since you're reading this tutorial, then you might not know all of the concepts used in the example, but don't worry, they will all be explained after the code listings:

```
<Window x:Class="WpfTutorialSamples.DataBinding.ConverterSample"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
                xmlns:local="clr-namespace:WpfTutorialSamples.DataBinding"
        Title="ConverterSample" Height="140" Width="250">
        <Window.Resources>
                <local:YesNoToBooleanConverter x:Key="YesNoToBooleanConverter" />
        </Window.Resources>
        <StackPanel Margin="10">
                <TextBox Name="txtValue" />
                <WrapPanel Margin="0,10">
                        <TextBlock Text="Current value is: " />
                        <TextBlock Text="{Binding ElementName=txtValue, Path=Text,
Converter={StaticResource YesNoToBooleanConverter}}"></TextBlock>
                </WrapPanel>
                <CheckBox IsChecked="{Binding ElementName=txtValue, Path=Text,
Converter={StaticResource YesNoToBooleanConverter}}" Content="Yes" />
        </StackPanel>
</Window>
```

```
using System;
using System.Windows;
using System.Windows.Data;

namespace WpfTutorialSamples.DataBinding
{
        public partial class ConverterSample : Window
        {
                public ConverterSample()
                {
                        InitializeComponent();
                }
        }

        public class YesNoToBooleanConverter : IValueConverter
        {
                public object Convert(object value, Type targetType, object parameter,
System.Globalization.CultureInfo culture)
                {
                        switch(value.ToString().ToLower())
                        {
                                case "yes":
                                case "oui":
                                        return true;
```

```
                                case "no":
                                case "non":
                                        return false;
                        }
                        return false;
                }

                public object ConvertBack(object value, Type targetType, object parameter,
System.Globalization.CultureInfo culture)
                {
                        if(value is bool)
                        {
                                if((bool)value == true)
                                        return "yes";
                                else
                                        return "no";
                        }
                        return "no";
                }
        }
}
```



Code-behind

So, let's start from the back and then work our way through the example. We have implemented a converter in the Code-behind file called YesNoToBooleanConverter. As advertised, it just implements the two required methods, called Convert() and ConvertBack(). **The Convert() methods assumes that it receives a string as the input (the *value* parameter) and then converts it to a Boolean true or false value, with a fallback value of false.** For fun, I added the possibility to do this conversion from French words as well.

The ConvertBack() method obviously does the opposite: It assumes an input value with a Boolean type and then returns the English word "yes" or "no" in return, with a fallback value of "no".

You may wonder about the additional parameters that these two methods take, but they're not needed in this example. We'll use them in one of the next chapters, where they will be explained.

XAML

In the XAML part of the program, we start off by declaring an instance of our converter as a resource for the window. We then have a TextBox, a couple of TextBlocks and a CheckBox control and this is where the interesting things are happening: We bind the value of the TextBox to the TextBlock and the CheckBox control and using the Converter property and our own converter reference, we juggle the values back and forth between a string and a Boolean value, depending on what's needed.

If you try to run this example, you will be able to change the value in two places: By writing "yes" in the TextBox (or any other value, if you want false) or by checking the CheckBox. No matter what you do, the change will be reflected in the other control as well as in the TextBlock.

Summary

This was an example of a simple value converter, made a bit longer than needed for illustrational purposes. In the next chapter we'll look into a more advanced example, but before you go out and write your own converter, you might want to check if WPF already includes one for the purpose. As of writing, there are more than 20 built-in converters that you may take advantage of, but you need to know their name.

WPF Übung Commands

For instance, if you have a typical interface with a main menu and a set of toolbars, an action like New or Open might be available in the menu, on the toolbar, in a context menu (e.g. when right clicking in the main application area) and from a keyboard shortcut like Ctrl+N and Ctrl+O.

Each of these actions needs to perform what is typically the exact same piece of code, so in a WinForms application, you would have to define an event for each of them and then call a common function. With the above example, that would lead to at least three event handlers and some code to handle the keyboard shortcut. Not an ideal situation.

Commands

With WPF, Microsoft is trying to remedy that with a concept called commands. It allows you to define actions in one place and then refer to them from all your user interface controls like menu items, toolbar buttons and so on. WPF will also listen for keyboard shortcuts and pass them along to the proper command, if any, making it the ideal way to offer keyboard shortcuts in an application.

Commands also solve another hassle when dealing with multiple entrances to the same function. In a WinForms application, you would be responsible for writing code that could disable user interface elements when the action was not available. For instance, if your application was able to use a clipboard command like Cut, but only when text was selected, you would have to manually enable and disable the main menu item, the toolbar button and the context menu item each time text selection changed.

With WPF commands, this is centralized. With one method you decide whether or not a given command can be executed, and then WPF toggles all the subscribing interface elements on or off automatically. This makes it so much easier to create a responsive and dynamic application!

Command bindings

Commands don't actually do anything by them self. At the root, they consist of the **ICommand** interface, which only defines an event and two methods: **Execute**() and **CanExecute**(). The first one is for performing the actual action, while the second one is for determining whether the action is currently available. To perform the actual action of the command, you need a link between the command and your code and this is where the CommandBinding comes into play.

A CommandBinding is usually defined on a Window or a UserControl, and holds a references to the Command that it handles, as well as the actual event handlers for dealing with the Execute() and CanExecute() events of the Command.

Pre-defined commands

You can of course implement your own commands, which we'll look into in one of the next chapters, but to make it easier for you, the WPF team has **defined over 100 commonly used commands** that you can use. They have been divided into **5 categories**, called **ApplicationCommands, NavigationCommands, MediaCommands, EditingCommands and ComponentCommands**. Especially ApplicationCommands contains commands for a lot of very frequently used actions like New, Open, Save and Cut, Copy and Paste.

Summary

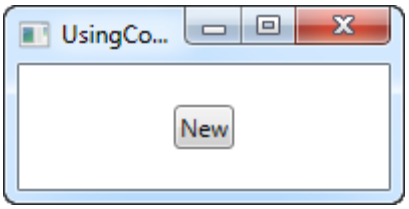
Commands help you to respond to a common action from several different sources, using a single event handler. It also makes it a lot easier to enable and disable user interface elements based on the current availability and state. This was all theory, but in the next chapters we'll discuss how commands are used and how you define your own custom commands.

Using WPF Commands -Bsp 1

We'll start off with a very simple example:

```
<Window x:Class="WpfTutorialSamples.Commands.UsingCommandsSample"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="UsingCommandsSample" Height="100" Width="200">
    <Window.CommandBindings>
        <CommandBinding Command="ApplicationCommands.New" Executed="NewCommand_Executed"
CanExecute="NewCommand_CanExecute" />
    </Window.CommandBindings>

    <StackPanel HorizontalAlignment="Center" VerticalAlignment="Center">
        <Button Command="ApplicationCommands.New">New</Button>
    </StackPanel>
</Window>
```

```
using System;
using System.Collections.Generic;
using System.Windows;
using System.Windows.Input;

namespace WpfTutorialSamples.Commands
{
        public partial class UsingCommandsSample : Window
        {
              public UsingCommandsSample()
              {
                        InitializeComponent();
              }

              private void NewCommand CanExecute(object sender, CanExecuteRoutedEventArgs e)
              {
                        e.CanExecute = true;
              }

              private void NewCommand Executed(object sender, ExecutedRoutedEventArgs e)
              {
                        MessageBox.Show("The New command was invoked");
              }
        }
}
```



We define a command binding on the Window, by adding it to its CommandBindings collection. We specify that Command that we wish to use (the New command from the ApplicationCommands), as well as two event handlers. The visual interface consists of a single button, which we attach the command to using the **Command** property.
In Code-behind, we handle the two events. The **CanExecute** handler, which WPF will call when the application is idle to see if the specific command is currently available, is very simple for this example, as we want this particular command to be available all the time. This is done by setting the **CanExecute** property of the event arguments to true.
The **Executed** handler simply shows a message box when the command is invoked. If you run the sample and press the button, you will see this message. A thing to notice is that this command has a default keyboard shortcut defined, which you get as an added bonus. Instead of clicking the button, you can try to press Ctrl+N on your keyboard - the result is the same.

Using the CanExecute method – Command Bsp2

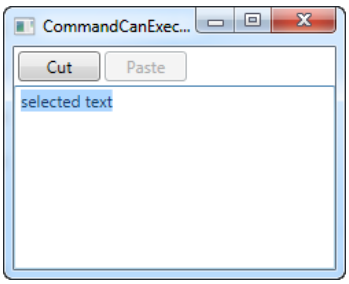
In the first example, we implemented a CanExecute event that simply returned true, so that the button would be available all the time. However, this is of course not true for all buttons - in many cases, you want the button to be enabled or disabled depending on some sort of state in your application.

A very common example of this is the toggling of buttons for using the Windows Clipboard, where you want the Cut and Copy buttons to be enabled only when text is selected, and the Paste button to only be enabled when text is present in the clipboard. This is exactly what we'll accomplish in this example:

```
<Window x:Class="WpfTutorialSamples.Commands.CommandCanExecuteSample"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="CommandCanExecuteSample" Height="200" Width="250">
    <Window.CommandBindings>
        <CommandBinding Command="ApplicationCommands.Cut" CanExecute="CutCommand CanExecute"
Executed="CutCommand Executed" />
        <CommandBinding Command="ApplicationCommands.Paste"
CanExecute="PasteCommand_CanExecute" Executed="PasteCommand_Executed" />
    </Window.CommandBindings>
    <DockPanel>
        <WrapPanel DockPanel.Dock="Top" Margin="3">
            <Button Command="ApplicationCommands.Cut" Width="60"> Cut</Button>
            <Button Command="ApplicationCommands.Paste" Width="60"
Margin="3,0">_Paste</Button>
        </WrapPanel>
        <TextBox AcceptsReturn="True" Name="txtEditor" />
    </DockPanel>
</Window>
```

```
using System;
using System.Collections.Generic;
using System.Windows;
using System.Windows.Input;

namespace WpfTutorialSamples.Commands
{
        public partial class CommandCanExecuteSample : Window
        {
           public CommandCanExecuteSample()
           {
                   InitializeComponent();
           }
```

```
            private void CutCommand_CanExecute(object sender, CanExecuteRoutedEventArgs e)
            {
                    e.CanExecute = (txtEditor != null) && (txtEditor.SelectionLength > 0);
            }

            private void CutCommand Executed(object sender, ExecutedRoutedEventArgs e)
            {
                     txtEditor.Cut();
            }

            private void PasteCommand_CanExecute(object sender, CanExecuteRoutedEventArgs e)
            {
                    e.CanExecute = Clipboard.ContainsText();
            }

            private void PasteCommand Executed(object sender, ExecutedRoutedEventArgs e)
            {
                    txtEditor.Paste();
            }
        }
}
```



So, we have this very simple interface with a couple of buttons and a TextBox control. The first button will cut to the clipboard and the second one will paste from it.

In Code-behind, we have two events for each button: One that performs the actual action, which name ends with _Executed, and then the CanExecute events. In each of them, you will see that I apply some logic to decide whether or not the action can be executed and then assign it to the return value **CanExecute** on the EventArgs.

The cool thing about this is that you don't have to call these methods to have your buttons updated - WPF does it automatically when the application has an idle moment, making sure that you interface remains updated all the time.

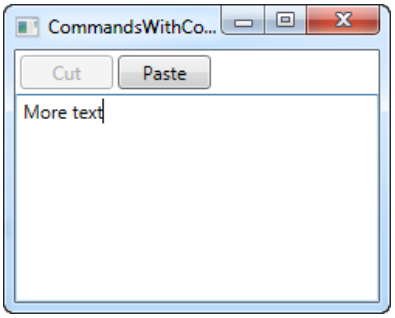
Default command behavior and CommandTarget -Command Bsp3

As we saw in the previous example, handling a set of commands can lead to quite a bit of code, with a lot of being method declarations and very standard logic. That's probably why the WPF team decided to handle some it for you. In fact, we could have avoided all of the Code-behind in the previous example, because a WPF TextBox can automatically handle common commands like Cut, Copy, Paste, Undo and Redo.

WPF does this by handling the Executed and CanExecute events for you, when a text input control like the TextBox has focus. You are free to override these events, which is basically what we did in the previous example, but if you just want the basic behavior, you can let WPF connect the commands and the TextBox control and do the work for you. Just see how much simpler this example is:

```
<Window x:Class="WpfTutorialSamples.Commands.CommandsWithCommandTargetSample"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="CommandsWithCommandTargetSample" Height="200" Width="250">
    <DockPanel>
        <WrapPanel DockPanel.Dock="Top" Margin="3">
            <Button Command="ApplicationCommands.Cut" CommandTarget="{Binding
ElementName=txtEditor}" Width="60">_Cut</Button>
```

```
            <Button Command="ApplicationCommands.Paste" CommandTarget="{Binding
ElementName=txtEditor}" Width="60" Margin="3,0">_Paste</Button>
        </WrapPanel>
        <TextBox AcceptsReturn="True" Name="txtEditor" />
    </DockPanel>
</Window>
```



No Code-behind code needed for this example - WPF deals with all of it for us, but only because we want to use these specific commands for this specific control. The TextBox does the work for us.

Notice how I use the **CommandTarget** properties on the buttons, to bind the commands to our TextBox control. This is required in this particular example, because the WrapPanel doesn't handle focus the same way e.g. a Toolbar or a Menu would, but it also makes pretty good sense to give the commands a target.

Summary

Dealing with commands is pretty straight forward, but does involve a bit extra markup and code. The reward is especially obvious when you need to invoke the same action from multiple places though, or when you use built-in commands that WPF can handle completely for you, as we saw in the last example.

Implementing a custom WPF Command Bsp4

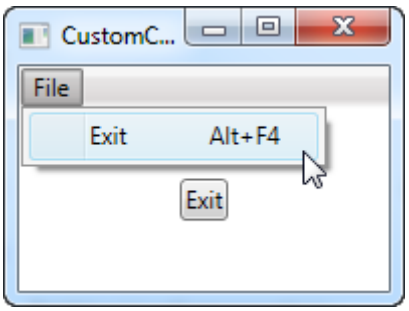
In the previous chapter, we looked at various ways of using commands already defined in WPF, but of course you can implement your own commands as well. It's pretty simply, and once you've done it, you can use your own commands just like the ones defined in WPF.

The easiest way to start implementing your own commands is to have a *static* class that will contain them. Each command is then added to this class as static fields, allowing you to use them in your application. Since WPF, for some strange reason, doesn't implement an Exit/Quit command, I decided to implement one for our custom commands example. It looks like this:

```
<Window x:Class="WpfTutorialSamples.Commands.CustomCommandSample"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:self="clr-namespace:WpfTutorialSamples.Commands"
        Title="CustomCommandSample" Height="150" Width="200">
    <Window.CommandBindings>
        <CommandBinding Command="self:CustomCommands.Exit"
CanExecute="ExitCommand CanExecute" Executed="ExitCommand Executed" />
    </Window.CommandBindings>
    <Grid>
        <Grid.RowDefinitions>
            <RowDefinition Height="Auto" />
            <RowDefinition Height="*" />
        </Grid.RowDefinitions>
        <Menu>
            <MenuItem Header="File">
                <MenuItem Command="self:CustomCommands.Exit" />
            </MenuItem>
        </Menu>
        <StackPanel Grid.Row="1" HorizontalAlignment="Center" VerticalAlignment="Center">
            <Button Command="self:CustomCommands.Exit">Exit</Button>
        </StackPanel>
```

```
    </Grid>
</Window>
```

```
using System;
using System.Collections.Generic;
using System.Windows;
using System.Windows.Input;

namespace WpfTutorialSamples.Commands
{
        public partial class CustomCommandSample : Window
        {
             public CustomCommandSample()
             {
                        InitializeComponent();
             }

             private void ExitCommand CanExecute(object sender, CanExecuteRoutedEventArgs e)
             {
                        e.CanExecute = true;
             }

             private void ExitCommand Executed(object sender, ExecutedRoutedEventArgs e)
             {
                        Application.Current.Shutdown();
             }
        }

        public static class CustomCommands
        {
                public static readonly RoutedUICommand Exit = new RoutedUICommand
                        (
                                "Exit",
                                "Exit",
                                typeof(CustomCommands),
                                new InputGestureCollection()
                                {
                                        new KeyGesture(Key.F4, ModifierKeys.Alt)
                                }
                        );

                //Define more commands here, just like the one above
        }
}
```



In the markup, I've defined a very simple interface with a menu and a button, both of them using our new, custom Exit command. This command is defined in Code-behind, in our own **CustomCommands** class, and then referenced in the CommandBindings collection of the window, where we assign the events that it should use to execute/check if it's allowed to execute.

All of this is just like the examples in the previous chapter, except for the fact that we're referencing the command from our own code (using the "self" namespace defined in the top) instead of a built-in command.

In Code-behind, we respond to the two events for our command: One event just allows the command to execute all the time, since that's usually true for an exit/quit command, and the other one calls the **Shutdown** method that will terminate our application. All very simple.

As already explained, we implement our Exit command as a field on a static CustomCommands class. There are several ways of defining and assigning properties on the commands, but I've chosen the more

compact approach (it would be even more compact if placed on the same line, but I've added line breaks here for readability) where I assign all of it through the constructor. The parameters are the text/label of the command, the name of the command, the owner type and then an InputGestureCollection, allowing me to define a default shortcut for the command (Alt+F4).

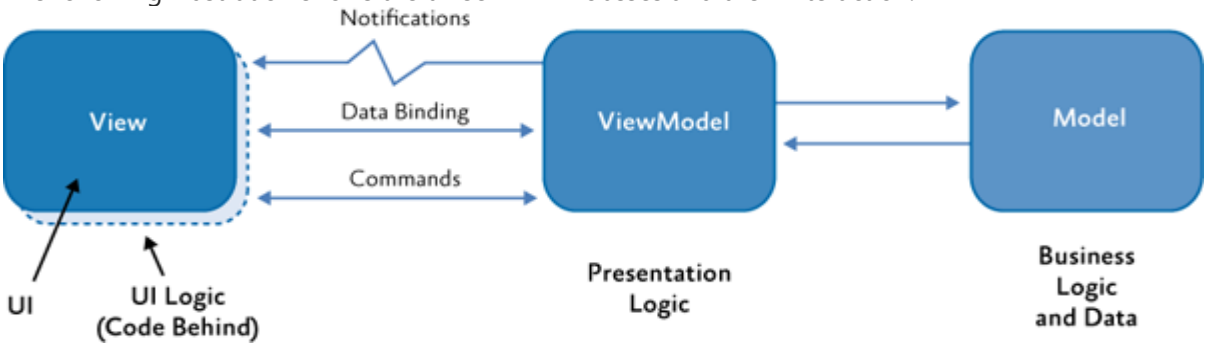
Summary

Implementing custom WPF commands is almost as easy as consuming the built-in commands, and it allows you to use commands for every purpose in your application. This makes it very easy to re-use actions in several places, as shown in the example of this chapter.

Implementing the MVVM Pattern – Command5

Class Responsibilities and Characteristics

In the MVVM pattern, the view encapsulates the UI and any UI logic, the view model encapsulates presentation logic and state, and the model encapsulates business logic and data. The view interacts with the view model through data binding, commands, and change notification events. The view model queries, observes, and coordinates updates to the model, converting, validating, and aggregating data as necessary for display in the view.
The following illustration shows the three MVVM classes and their interaction.



The View Class

To summarize, the view has the following key characteristics:

- The view is a **visual element**, such as a window, page, user control, or data template. The view defines the controls contained in the view and their visual layout and styling.
- The view **references the view model through** its **DataContext** property. The **controls** in the view are **data bound** to the properties and **commands** exposed by the view model.
- The **view may customize** the data binding behavior between the view and the view model. For example, the view may use value converters **to format the data** to be displayed in the UI, or it may use validation rules to provide additional input data validation to the user.
- The **view defines** and handles UI visual behavior, **such as animations** or transitions that may be triggered from a state change in the view model or via the user's interaction with the UI.
- The view's **code-behind** may **define UI logic** to implement visual behavior that is difficult to express in XAML or that requires direct references to the specific UI controls defined in the view.

The View Model Class

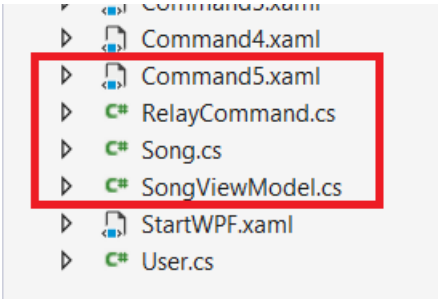
- The view model is a **non-visual class and does not derive from any WPF base class**. It **encapsulates the presentation logic** required to support a use case or user task in the application. The view model is testable independently of the view and the model.
- The view model typically **does not directly reference the view.** It **implements properties and commands** to which the view can data bind. **It notifies the view** of any state changes via change **notification events via** the **INotifyPropertyChanged** and **INotifyCollectionChanged** interfaces.
- The view model coordinates the view's interaction with the model. It may convert or manipulate data so that it can be easily consumed by the view and may implement additional properties that may not be present on the model. It may **also implement data validation via** the **IDataErrorInfo** or **INotifyDataErrorInfo** interfaces.
- The view model may define logical states that the view can represent visually to the user.

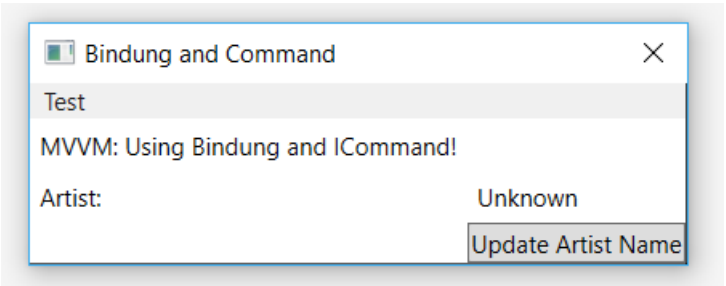
The Model Class

- Model classes are **non-visual classes** that **encapsulate the application's data and business logic**. They are responsible for managing the application's data and for ensuring its consistency and validity by encapsulating the required business rules and data validation logic.
- **The model classes do not directly reference the view or view model classes** and have no dependency on how they are implemented.
- The model classes typically provide property and collection change notification events through the **INotifyPropertyChanged** and **INotifyCollectionChanged** interfaces. This allows them to be easily data bound in the view. Model classes that represent collections of objects typically derive from the **ObservableCollection<T>** class.
- The model classes typically **provide data validation** and error reporting through either the **IDataErrorInfo** or **INotifyDataErrorInfo** interfaces.
- The model classes are typically used in conjunction with a service or repository that encapsulates data access and caching.

Sample:

- View: Command5.xaml
- ViewModel: SongViewModel.cs
- Model: Model.cs
- HelperClass: RelayCommand.cs





File: Command5.xaml:

```
<Window x:Class="WPF_Bindings.Command5"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:local="clr-namespace:WPF_Bindings"
        Title="Binding and Command" Height="350" Width="525" SizeToContent="WidthAndHeight"
ResizeMode="NoResize">
    <Window.DataContext>
        <!-- Declaratively create an instance of our SongViewModel -->
        <local:SongViewModel />
    </Window.DataContext>
    <Grid>
        <Grid.RowDefinitions>
            <RowDefinition Height="Auto" />
            <RowDefinition Height="Auto" />
            <RowDefinition Height="Auto" />
            <RowDefinition Height="Auto" />
        </Grid.RowDefinitions>
        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="Auto" />
            <ColumnDefinition Width="Auto" />
        </Grid.ColumnDefinitions>
        <Menu Grid.Row="0" Grid.ColumnSpan="3">
            <MenuItem Header="Test">
                <MenuItem Header="Update Artist" Command="{Binding UpdateArtistName}" />
            </MenuItem>
        </Menu>
        <Label Grid.Column="0" Grid.Row="1" Content="MVVM: Using Bindung and ICommand!" />
        <Label Grid.Column="0" Grid.Row="2" Content="Artist:  " />
        <Label Grid.Column="1" Grid.Row="2" Content="{Binding ArtistName}" />
        <Button Grid.Column="1" Grid.Row="3" Name="ButtonUpdateArtist" Content="Update
Artist Name" Command="{Binding UpdateArtistName}" />
    </Grid>
</Window>
```

File: SongViewModel.cs

```
public class SongViewModel : INotifyPropertyChanged
    {
        #region Construction
        public SongViewModel()
        {
            _song = new Song { ArtistName = "Unknown", SongTitle = "Unknown" };
        }
        #endregion

        #region Members
        Song _song;
        int _count = 0;
        #endregion

        #region Properties
        public Song Song
        {
            get
            {
                return _song;
            }
            set
            {
                _song = value;
            }
        }
```

```
        public string ArtistName
        {
            get { return Song.ArtistName; }
            set
            {
                if (Song.ArtistName != value)
                {
                    Song.ArtistName = value;
                    RaisePropertyChanged("ArtistName");
                }
            }
        }
        #endregion

        #region INotifyPropertyChanged Members

        public event PropertyChangedEventHandler PropertyChanged;

        #endregion

        #region Methods

        private void RaisePropertyChanged(string propertyName)
        {
            // take a copy to prevent thread issues
            PropertyChangedEventHandler handler = PropertyChanged;
            if (handler != null)
            {
                handler(this, new PropertyChangedEventArgs(propertyName));
            }
        }
        #endregion

        #region Commands
        void UpdateArtistNameExecute()
        {
            ++_count;
            ArtistName = string.Format("Elvis ({0})", _count);
        }

        bool CanUpdateArtistNameExecute()
        {
            return true;
        }

        public ICommand UpdateArtistName { get { return new
RelayCommand(UpdateArtistNameExecute, CanUpdateArtistNameExecute); } }
        #endregion
    }
```

File: Song.cs

```
public class Song
    {
        #region Members
        string _artistName;
        string _songTitle;
        #endregion


        #region Properties
        /// <summary>
        /// The artist name.
        /// </summary>
        public string ArtistName
```

```
        {
            get { return _artistName; }
            set { _artistName = value; }
        }

        /// <summary>
        /// The song title.
        /// </summary>
        public string SongTitle
        {
            get { return _songTitle; }
            set { _songTitle = value; }
        }

        #endregion
    }
```

File: RelayCommand.cs

```
public class RelayCommand : ICommand
    {
        readonly Func<Boolean> _canExecute;
        readonly Action _execute;

        public RelayCommand(Action execute) : this(execute, null) { }

        public RelayCommand(Action execute, Func<Boolean> canExecute)
        {
            if (execute == null)
                throw new ArgumentNullException("execute");
            _execute = execute;
            _canExecute = canExecute;
        }


        public event EventHandler CanExecuteChanged
        {
            add
            {
                if (_canExecute != null)
                    CommandManager.RequerySuggested += value;
            }

            remove
            {
                if (_canExecute != null)
                    CommandManager.RequerySuggested -= value;
            }
        }

        [DebuggerStepThrough]
        public Boolean CanExecute(Object parameter)
        {
            return _canExecute == null ? true : _canExecute();
        }

        public void Execute(Object parameter)
        {
            _execute();
        }
    }
```