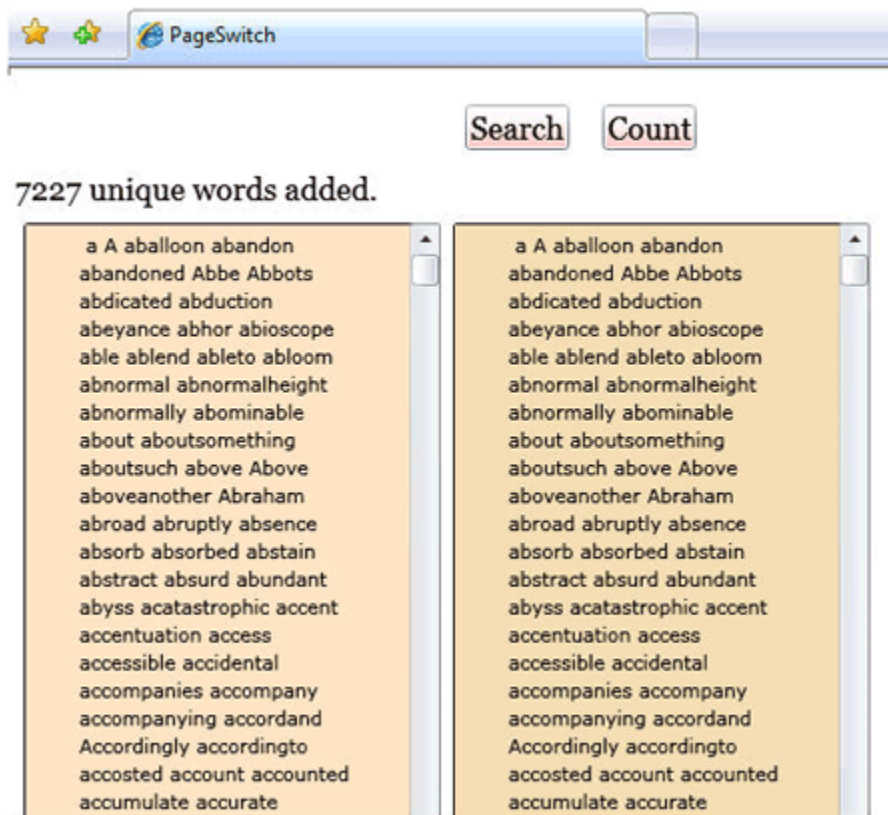


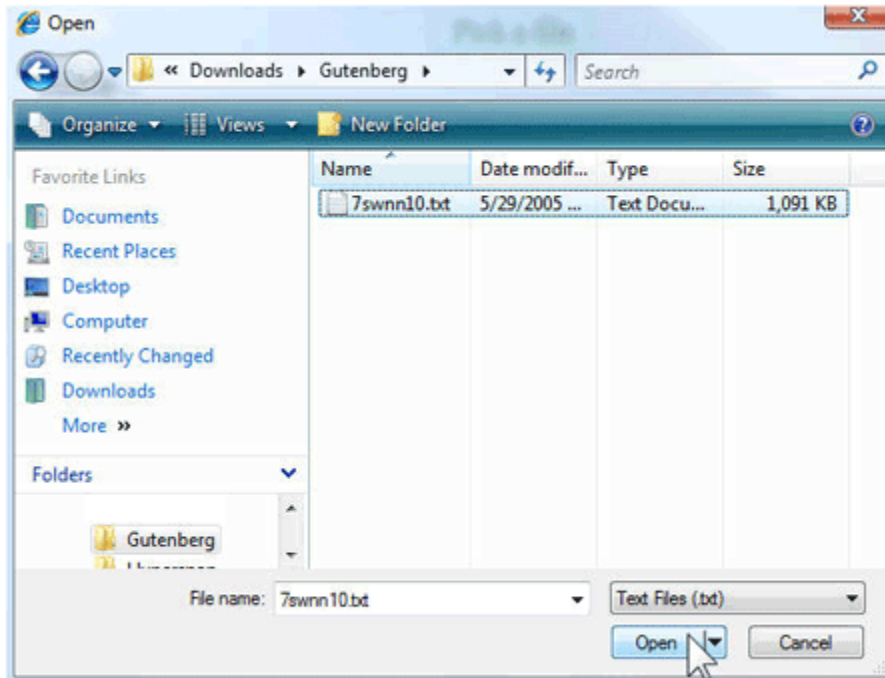
# One Application – Many Topics

The primary focus of this tutorial is on building an application that allows you to switch from one page to another, passing data from the first to the second. To demonstrate the usefulness of this, we'll create a list of many words in the first page, and pass them to one of two other pages: *Search* or *Count*. The Search page will use the list as the source for an AutoComplete box, the Count page will use the words as the data for a graph as shown here:



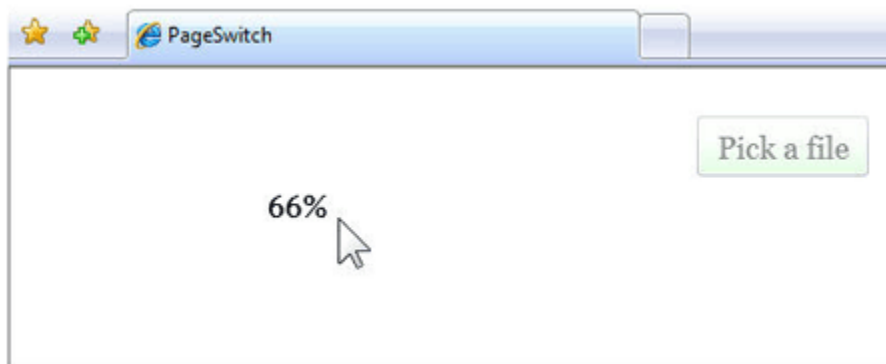
**Figure 8-1. First Page View**

To fill the list we'll look at the Open File dialog box,



**Figure 8-2. Open File Dialog Box**

and to ensure that we keep the user informed as to progress, we'll look at using a worker-thread.



**Figure 8-3. Progress**

In the second part of the tutorial (Tutorial 9) we'll optimize performance by caching the words using Isolated Storage.

Note that while this tutorial will show how to connect the AutoCompleteBox and the Chart, both of which are from the Silverlight Control Toolkit, it will not endeavor to explore these controls in depth; a full exploration would be lengthy and will be reserved for another tutorial, videos and blog entries.

# The Architecture of Multi-Page Applications

There are a number of ways of implementing a multi-page application. The architecture we're going to look at was initially suggested by Ashish Shetty of the development team and was then modified by a reader, Lucas Stark (Senior Web Developer at Delta College) and then modified once more by myself. It is not the only way to do this, and indeed there are commercial libraries that offer much more complex and multi-featured approaches. But this is an approach that works, that is robust, and that illustrates many interesting aspects of the Silverlight model.

We begin by noting that every "page" in a Silverlight application is actually of type `UserControl` and that one `UserControl` can contain another as its "Contents." This is true throughout Silverlight.

## Creating The First Solution

Let's begin by creating a new Silverlight application in Visual Studio (allowing it to create an ASP.NET Web Application Project) named *PageSwitchSimple*.

Notice that Visual Studio creates one page, *Page.xaml* for you by default and that *Page.Xaml* is a `UserControl` (you can see that by looking at the Xaml view of *Page.xaml*).

### Key Files

Our architecture for switching pages requires four files that must be added to any project that wants to participate in this approach:

1. A user control named *PageSwitcher.xaml* and its code-behind file *PageSwitcher.xaml.cs*
2. A static Class named *Switicher*
3. An Interface: *ISwitchable*

In addition, four lines must be added to *App.xaml.cs* as will be shown later.

Much of the rest of this tutorial will explore how these four files are implemented, how they fit together and how they allow all the other pages in your application to switch from one to another, passing data from one to the other in a natural and seamless manner.

## PageSwitcher.xaml

All the work that follows will be done in the main project (the one with *Page.xaml* and *App.xaml*).

Add *PageSwitcher.xaml* as you would any other `UserControl`, but once it is created, take out the grid that Visual Studio creates for you and change the width and height of the `UserControl` to 800 x 600 (these values are arbitrarily large,

```

<UserControl x:Class="PageSwitchSimple.PageSwitcher"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Width="800" Height="600">

</UserControl>

```

This is the one UserControl that you will leave empty, because we will fill it programmatically with the contents of other pages as we go. It is, in essence, the vessel into which we'll be pouring each page we want to view.

The core of PageSwitcher.xaml.cs is an overloaded Navigate method, that takes either a UserControl or a UserControl and an object.

```

using System;
using System.Windows.Controls;

namespace PageSwitchSimple
{
    public partial class PageSwitcher : UserControl
    {
        public PageSwitcher()
        {
            InitializeComponent();
        }

        // 1st overload
        public void Navigate( UserControl nextPage )
        {
            this.Content = nextPage;
        }

        // 2nd overload
        public void Navigate( UserControl nextPage, object state )
        {
            this.Content = nextPage;
            ISwitchable s = nextPage as ISwitchable;
            if ( s != null )
            {
                s.UtilizeState( state );
            }
            else
            {
                throw new ArgumentException( "nextPage is not ISwitchable! "
                    + nextPage.Name.ToString() );
            }
        }
    }
}

```

The first overload of `Navigate` sets the content of *PageSwitcher* to whatever UserControl you pass in. Since you'll pass in a page, the effect, from the point of view of the user, is that the page you pass in becomes the current page.

## Testing The Premise

To see this, we'll create a bit of temporary code. First, comment out the second overload.

Second, add to the constructor, the following line of code:

```
this.Content = new Page2();
```

Finally, open App.xaml.cs and locate the method Application\_Startup. Change the assignment of RootVisual from new Page() to new PageSwitcher() and run the application. You should see Page2 displayed.

By running the application in the debugger and placing a break point on the Application\_Startup method you can see that what is actually going on is that PageSwitcher is being created and in its constructor it is filling its contents with a new instance of Page2.

This demonstrates that our premise is correct; when PageSwitcher fills its contents with a UserControl it looks as if that UserControl is all that is being displayed.

## The ISwitchable Interface

Great. You can now delete the line you added to PageSwitcher's constructor. Before you can uncomment the second overload of Navigate, however, you need to create the interface ISwitchable as it is used in that overload,

```
public void Navigate( UserControl nextPage, object state )
{
    this.Content = nextPage;
    ISwitchable s = nextPage as ISwitchable;
}
```

The interface is just a promise that any class implementing the interface will implement UtilizeState. UtilizeState is a method that takes an object and returns void,

```
public interface ISwitchable
{
    void UtilizeState( object state );
}
```

This interface is critical, however; it allows us to assume that if you pass in an object when you navigate to another page, we can tell that page to use that object because that page will be Switchable and will know what to do with that object.

You create the interface by telling visual studio you want to create a new class named ISwitchable.cs and then replacing the class that it creates for you with the interface code shown.

Once ISwitchable has been declared, you can uncomment the second Navigate method in SearchPage.xaml.cs.

## How Does The Page Get the PageSwitcher Instance?

For one page to navigate to another it must call the Navigate method on PageSwitcher. There are two approaches. The approach that I took initially was to have the Page ask for its parent, which returns the PageSwitcher instance (since the page is actually the contents of the PageSwitcher). This is complex and asks the Page to know more about the architecture than is appropriate, and it repeats the acquire and cast code through every page.

A more elegant solution is to use an intermediary: a static class named Switcher that will be initialized with an instance of PageSwitcher and that will provide every page with a one line access to navigation. This is much more elegant, much easier to maintain and to scale, and does a much better job of encapsulating responsibility.

Switcher has two overloaded versions of its static Switch method; as you might expect, one that takes a UserControl, and one that takes a UserControl and an object. Stripped of its error checking it looks like this:

```
public static class Switcher
{
    public static PageSwitcher pageSwitcher;

    public static void Switch( UserControl newPage )
    {
        pageSwitcher.Navigate( newPage );
    }

    public static void Switch( UserControl newPage, object state )
    {
        pageSwitcher.Navigate( newPage, state );
    }
}
```

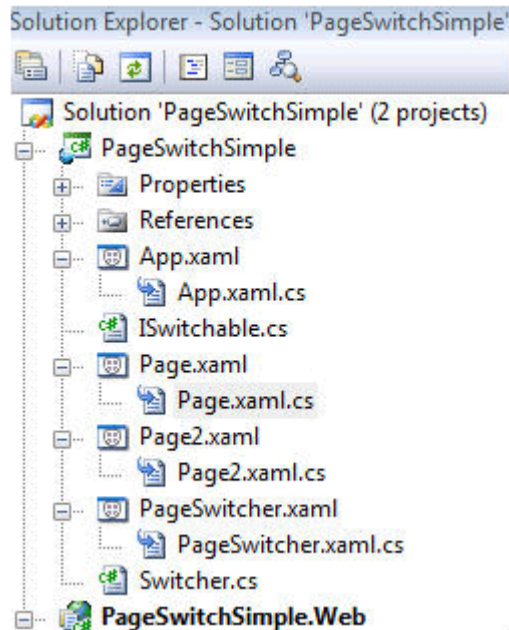
You can see that its job is to hold onto the pageSwitcher class and then just invoke that instance's Navigate method. This greatly simplifies the switching for each page. The code to switch to the Search page and pass in a collection of words becomes just this line:

```
Switcher.Switch ( new Search(), words );
```

The entire mechanism of the manipulation of the PageSwitcher UserControl's contents and of the passing of the object are appropriately hidden from the calling class, and because Switch is a static method, no instance of Switcher needs to be created. Very nice.

### Synchronization Check Point

Right now your PageSwitchSimple solution should have two projects, one of which is called PageSwitchSimple and the other PageSwitchSimple.web. PageSwitchSimple should look like this:



**Figure 8-4. PageSwitchSimple**

Before we go any further, we must update App.xaml.cs. Please replace the contents of Application\_Startup with the following four lines of code,

```
PageSwitcher pageSwitcher = new PageSwitcher();
this.RootVisual = pageSwitcher;
Switcher.pageSwitcher = pageSwitcher;
Switcher.Switch( new Page() );
```

As you can see, this creates an instance of PageSwitcher and sets it as the value for RootVisual (which can only be set once at start up and not reset while the program is running), and also sets the static pageSwitcher property to the same value. Finally, the fourth line calls the static method Switch passing in a new instance of Page, invoking the switching mechanism to display the first page.

Let's create a very simple body for both Page and Page2 to test the mechanism we've created.

In Page, add the following Xaml inside the grid,

```
<TextBlock Text="Your Name: " FontSize="18" />
<TextBox x:Name="Name" FontSize="18" Width="150" Height="35"
VerticalAlignment="Top" Margin="5"/>
<Button x:Name="ChangePage" Content="Change" FontSize="18"
Width="100" Height="50" />
```

This creates a prompt, a textbox to fill in and a button to click to change pages.

In the supporting code in Page.xaml.cs you only need to register and then implement the button's event handler, in which you'll pick up the text from the textbox and pass it to a new instance of Page2. You'll navigate to this new instance of Page2 through the static Switch method of the Switcher class,

```
public Page()
{
    InitializeComponent();
    ChangePage.Click += new RoutedEventHandler( ChangePage_Click );
}

void ChangePage_Click( object sender, RoutedEventArgs e )
{
    Switcher.Switch( new Page2(), Name.Text );
}
```

In truth, this comes down to one line of interesting code, the invocation of the static Switch method, passing in the new Page2 and the text. Before we trace how this works, let's create Page2. In Page2.xaml we'll add a text box to display whatever message is sent in from Page and a button to return to the first page,

```
<TextBlock x:Name="Message" Text="Page2" FontSize="18" />
<Button x:Name="ChangePage" Content="Change" FontSize="18"
Width="100" Height="50" />
```

To distinguish the two pages, modify the Grid in Page2 to have a background color of Bisque,

```
<Grid x:Name="LayoutRoot" Background="Bisque">
```

The code support for Page2 is to have it implement the ISwitchable interface, which requires two steps: declaring that you support the interface,

```
public partial class Page2 : UserControl, ISwitchable
```

and then implementing the UtilizeState method. This is where you might have a moment's hesitation, but thinking it through, the state you are being passed is the text from the text box on Page, and so your goal is to display it in your message TextBlock,

```
public void UtilizeState( object state )
{
    Message.Text = state.ToString();
}
```

The only other code in this file is to declare the event handler for the button,

```
public Page2()
{
    InitializeComponent();
```

```

    ChangePage.Click += new RoutedEventHandler( ChangePage_Click );
}

```

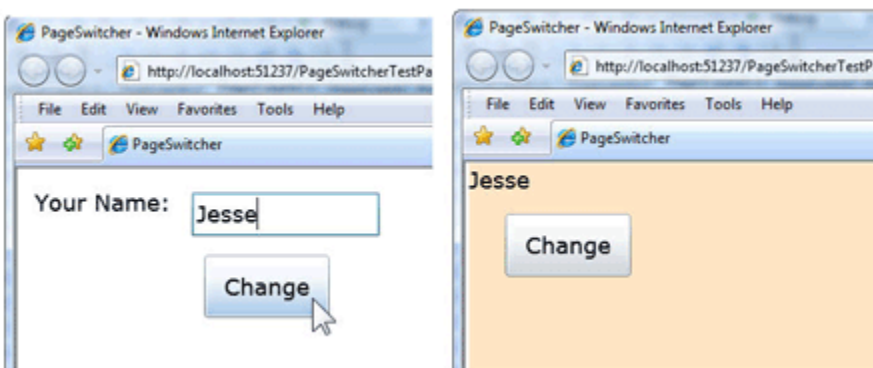
And to implement that. In this case, you have no data to pass back, so you can invoke the simpler overload,

```

void ChangePage_Click( object sender, RoutedEventArgs e )
{
    Switcher.Switch( new Page() );
}

```

That's it! You now have a working program that switches pages. The first page will come up and ask you to fill in your name, and when you do the second page will display the name,



**Figure 8-5. Page Switching Test 2**

## Walk Through The Details

Before going on, it is critical that you are comfortable with the steps that occur when you click the Change button on the first page. I strongly suggest not only creating (or downloading) this application and running it, but stepping through it in the debugger. Here's a fast review, but nothing substitutes for stepping through it yourself.

### Application Startup

When the application begins `Application_Startup` is invoked automatically. As noted above, a new instance of `PageSwitcher` is created and stashed away in `Switcher` (and as the `RootVisual`). Then a new instance of `Page` is created, and `Switch` is called, with control passing to that static method.

The `Switch` method is overloaded, and `Application_Startup` has called the first version that takes only one argument. It looks for (and finds!) the `pageSwitcher` that was just created, and calls `Navigate`, passing along the `UserControl` that it received.

Now would be a good time to revise Switcher to add the error checking that we stripped out earlier. The full version of Switcher.cs is shown here,

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

namespace PageSwitchSimple
{
    public static class Switcher
    {
        public static PageSwitcher pageSwitcher;

        public static void Switch( UserControl newPage )
        {
            if ( pageSwitcher != null )
            {
                pageSwitcher.Navigate( newPage );
            }
            else
            {
                throw new Exception( "Switcher.pageSwitcher is null" );
            }
        }

        public static void Switch( UserControl newPage, object state )
        {
            if ( pageSwitcher != null )
            {
                pageSwitcher.Navigate( newPage, state );
            }
            else
            {
                throw new Exception( "Switcher.pageSwitcher is null" );
            }
        }
    }
}
```

Control switches to the PageSwitcher instance's overloaded Navigate method, where the content is set,

```
public void Navigate( UserControl nextPage )
{
    this.Content = nextPage;
}
```

At this point, the first page is displayed and the system is quiescent.

## Clicking the Change Button

When you click the button, the button handler is invoked,

```
Switcher.Switch( new Page2(), Name.Text );
```

Control again passes to the static Switch method, but this time the second overload, taking both a UserControl and an object. Since Object is the root of all classes, it happily accepts the string that we pass.

Notice that a new instance of Page2() is created. You'll see that this replaces what was in the contents of the PageSwitcher object (Page) and thus the old contents are destroyed. We are parsimonious with the user's memory.

Switch has a bit more work to do now. It sets its contents, but then it needs to call UtilizeState on the new page, passing in the state it was given. Unfortunately, it can't call this method on a UserControl, so it must cast, testing to ensure that the UserControl does in fact implement ISwitchable.

The as operator will return an instance of a class that implements the interface if that class does implement the interface, or it will return null if the class does not.

```
ISwitchable s = nextPage as ISwitchable;  
if ( s != null )  
{  
    s.UtilizeState( state );  
}
```

The object s is referred to as an instance of type ISwitchable, but of course that is just shorthand for "s is an instance of a class that implements ISwitchable." In any case, s can now call the ISwitchable method UtilizeState, which it does, passing in the state object it received.

As an aside, note that it is not possible to instantiate an Interface object in C#. Thus you could not write

```
ISwitchable s = new ISwitchable();
```

What we are doing here is just creating a reference that is typed as "class that implements this interface" and pointing that reference to an object that already exists.

This invocation of UtilizeState by the Switcher static method obviates the need for either UserControl to specifically request the UserState and thus for either UserControl to even be aware of the existence of PageSwitcher (sweet!).

Also notice that PageSwitcher is just a courier, it never looks inside the object, it has no idea what type it is, and it doesn't have to store it; PageSwitcher just accepts the object from the calling page and passes it to the UtilizeState method of the called page.

UtilizeState in this case, assigns the object passed in to the Text property of the Message textBlock. Since we know the state object is a string, we are free to make this assignment, but we still have to either cast it or call ToString,

```
public void UtilizeState( object state )
{
    Message.Text = state.ToString();
}
```

The order of operations is critical here, but entirely automatic. Page2 is created on the call to Switch, thus its constructor is called and its components initialized *before* PageSwitcher tries to call its UtilizeState method (good thing, too, as otherwise the method would fail), however, UtilizeState is called before PageSwitcher displays its new contents, so the state has been utilized (the message has been set) in time for display to the user.

## A More Practical Use

Now that we have the fundamental architecture in place, we have only to create useful pages to switch among. We'll create three, as mentioned above. The first will generate a large list of words that the other two will use.

I'll begin by creating a new solution named PageSwitching. Once this is open in Visual Studio, I'll copy the following files from PageSwitch into the new directory: App.xaml.cs, ISwitchable.cs, PageSwitcher.xaml, PageSwitcher.xaml.cs and Switcher.cs.

App.xaml.cs will overwrite the one created by Visual Studio (make sure you adjust the namespace in the file), the rest are new, and so you need to add them to your project by right clicking on the project in the Solution Explorer and choosing Add Existing Item,

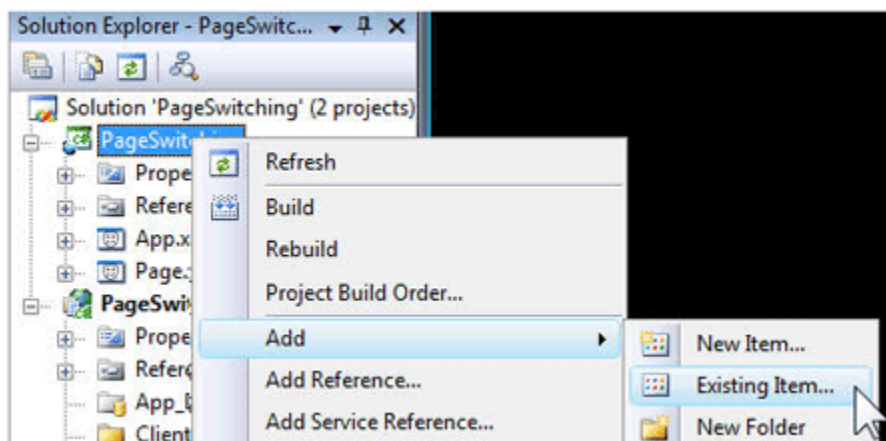
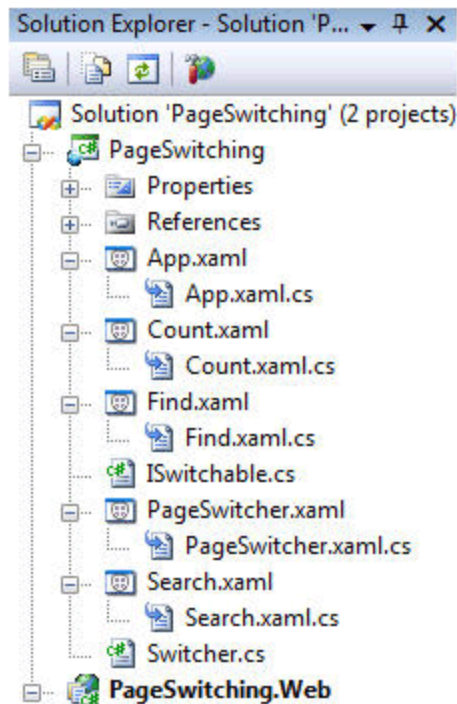


Figure 8-6. Adding Existing Items

Delete Page.xaml and Page.xaml.cs and add three new UserControls: Find, Search and Count. When you're done, your project should look like this:



**Figure 8-7. Project Page Switching**

Let's add the layout for the Find page; the first page that will be used to create the list of words.

Here is the Xaml I used for laying out the page, feel free to adjust it in any way you find aesthetically pleasing,

```
<UserControl x:Class="PageSwitching.Find"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Width="500" Height="600">
  <Grid
    x:Name="LayoutRoot"
    Background="White">
    <Grid.RowDefinitions>
      <RowDefinition
        Height="0.078*" />
      <RowDefinition Height="0.072*" />
      <RowDefinition
        Height="0.85*" />
    </Grid.RowDefinitions>
    <Grid.ColumnDefinitions>
      <ColumnDefinition
        Width="0.5*" />
      <ColumnDefinition
        Width="0.5*" />
    </Grid.ColumnDefinitions>
  </Grid>
</UserControl>
```

```

<TextBlock
  x:Name="Message"
  Text="Ready..."
  TextWrapping="Wrap"
  FontFamily="Georgia"
  FontSize="18"
  VerticalAlignment="Bottom"
  HorizontalAlignment="Left"
  Visibility="Visible" Grid.Row="1" />
<ScrollView
  x:Name="WordDisplayViewer"
  BorderBrush="Black"
  BorderThickness="1"
  Grid.Row="2"
  Grid.Column="0"
  Margin="5,10,0,2"
  Background="Bisque"
  VerticalScrollBarVisibility="Auto"
  HorizontalScrollBarVisibility="Hidden"
  VerticalAlignment="Stretch"
  Width="240"
  HorizontalAlignment="Left"
  Visibility="Visible">
  <TextBlock
    x:Name="WordDisplay"
    TextWrapping="Wrap"
    Text="WordDisplay"
    Width="160" />
</ScrollView>
<ScrollView
  x:Name="SortDisplayViewer"
  BorderBrush="Black"
  BorderThickness="1"
  Grid.Row="2"
  Grid.Column="1"
  Margin="0,10,8,0"
  Width="240"
  Background="Wheat"
  VerticalScrollBarVisibility="Auto"
  HorizontalScrollBarVisibility="Hidden"
  VerticalAlignment="Stretch"
  HorizontalAlignment="Right"
  Visibility="Visible">
  <TextBlock
    x:Name="SortDisplay"
    TextWrapping="Wrap"
    Width="160"
    Text="SortDisplay" />
</ScrollView>
<StackPanel
  Height="Auto"
  VerticalAlignment="Stretch"
  Grid.Column="1"
  Orientation="Horizontal" Grid.RowSpan="2">
  <Button
    x:Name="FilePicker"
    Content="Pick a file"

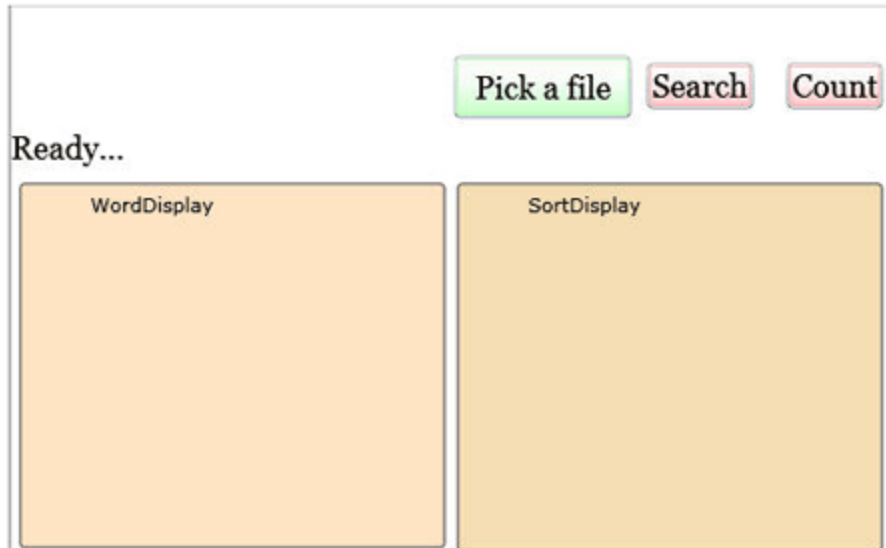
```

```

        Width="100"
        Background="#FF00FF00"
        FontFamily="Georgia"
        FontSize="18"
        Height="35" HorizontalAlignment="Center"
VerticalAlignment="Center" />
    <Grid Height="Auto" x:Name="ButtonGrid" Width="153">
        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="0.5*" />
            <ColumnDefinition Width="0.5*" />
        </Grid.ColumnDefinitions>
        <Grid.RowDefinitions>
            <RowDefinition Height="0.5*" />
            <RowDefinition Height="0.5*" />
        </Grid.RowDefinitions>
        <Button
x:Name="SearchPage"
Content="Search"
Background="#FFFF0000"
FontFamily="Georgia"
FontSize="18"
Margin="5,0,5,0"
Visibility="Visible"
    HorizontalAlignment="Center"
    VerticalAlignment="Center"
    Grid.ColumnSpan="1"
    Grid.RowSpan="2" />
        <Button
x:Name="CountPage"
Content="Count"
Background="#FFFF0000"
FontFamily="Georgia"
FontSize="18"
Visibility="Visible"
    VerticalAlignment="Center"
    Grid.Row="0"
    Grid.Column="1"
    HorizontalAlignment="Center"
    Grid.RowSpan="2" />
    </Grid>
</StackPanel>
</Grid>
</UserControl>

```

Here's what it looks like (slightly cropped)



**Figure 8-8. Find Page**

Find.xaml.cs must implement event handlers for the three buttons,

```
public Find()
{
    InitializeComponent();
    FilePicker.Click += new RoutedEventHandler( FilePicker_Click );
    SearchPage.Click += new RoutedEventHandler( ChangePage );
    CountPage.Click +=new RoutedEventHandler( ChangePage );
}
```

Notice that the SearchPage and CountPage buttons will share an event handler; this is because their implementations are very similar,

```
void ChangePage( object sender, RoutedEventArgs e )
{
    Button b = e.OriginalSource as Button;
    string btnName = b.Content.ToString().ToUpper();
    if ( btnName == "SEARCH" )
        Switcher.Switch( new SearchPage(), SortedWords );
    else
        Switcher.Switch( new CountPage(), SortedWords );
}
```

The logic is to cast the OriginalSource property of the Event Argument to type Button and then to use that to extract the uppercase form of the contents of the button. We then compare that with the word search, if they match, we call the Switch method for the SearchPage (otherwise for the CountPage) and pass in SortedWords, which is a collection of the words the new page will need.

We collect those words by reading through a large document and picking out all the unique words. A good way to get a document for this is from Project Gutenberg which supplies free e-books many of which are in the public domain. For this tutorial I'll use Proust's Swann's Way

(<http://www.gutenberg.org/etext/7178>) which is in the public domain in the United States [please check the laws of your own country before using this document].

I've downloaded that file and placed it in a directory on my disk. When the user clicks on the Pick A File button, the event handler opens the dialog and asks the user to pick a file,

```
void FilePicker_Click( object sender, RoutedEventArgs e )
{
    FilePicker.IsEnabled = false;
    OpenFileDialog openFileDialog1 = new OpenFileDialog();
    openFileDialog1.Filter = "Text Files (.txt)|*.txt|All Files (*.*)|*.*";
    openFileDialog1.FilterIndex = 1;
    openFileDialog1.Multiselect = false;
    bool? userClickedOK = openFileDialog1.ShowDialog();
}
```

If the user clicks on a file and then the OK button, the userClickedOK nullable-boolean will have the value true. At that point you want to open a StreamReader on the file and read its contents into a StringBuilder, reading all the way to the end of the file, adding each line to your StringBuilder,

```
if ( userClickedOK == true )
{
    // limit how much you read to save time
    const long MAXBYTES = 2000; // 200000
    System.IO.FileInfo file = openFileDialog1.File;
    StringBuilder sb = new StringBuilder();

    if ( file != null )
    {
        System.IO.Stream fileStream = file.OpenRead();
        using ( System.IO.StreamReader reader =
            new System.IO.StreamReader( fileStream ) )
        {
            string temp = string.Empty;

            // append to the string builder while you have lines to
            // read and have not hit the MAXBYTES limit
            try
            {
                do
                {
                    temp = reader.ReadLine();
                    sb.Append( temp );
                } while ( temp != null && sb.Length < MAXBYTES );
            }
            catch { } // for now, ignore exceptions
        }
        fileStream.Close(); // tidy up (should be in a finally block)
    }
}
```

## From a String, an Array of Words

With all the lines in the `StringBuilder` (`sb`) you are ready to break the string into words; which we'll define as characters separated by white space. We'll do this with the `Split` method of the `RegularExpression` class; which will return an array of all the words,

```
string pattern = "\\b";
string[] allWords = System.Text.RegularExpressions.Regex.Split(
    sb.ToString(), pattern );
```

If you are not familiar with Regular Expressions at all, I recommend two resources:

*RegExBuddy* (<http://www.regexbuddy.com/>) is a wonderful interactive tool that both helps you create regular expressions and learn about them.

The book *Mastering Regular Expressions* (3rd Edition) (<http://www.tinyurl.com/MasteringRE>) by Jeffrey Friedl is the best I know on the subject.

If you are familiar with Regular expressions but not the Regular expressions object in C#, then I recommend a good book on C#, including I'm pleased to say my most recent book, *Learning C# 3.0* which is released despite the fact that as of this writing, Amazon doesn't know that.

We'll define a member variable to hold our finished list of unique words,

```
private List<string> words = new List<string>();
```

and we'll add two properties to get back either the words in the order they were added, or sorted alphabetically,

```
public List<string> Words
{
    get { return this.words; }
}

public List<string> SortedWords
{
    get
    {
        List<string> temp = this.words;
        temp.Sort();
        return temp;
    }
}
```

## Creating the Unique List of Words

We can now iterate through the array returned by the regular expression and check each word to make sure it is not already in our collection (thus ensuring we have each word only once), that it

is not of length 0 (which should not be possible) and that it is not “junk” where junk is defined as containing punctuation, digits, symbols or separators. We’ll delegate the junk-detection to a helper method, `IsJunk`:

```
foreach ( string word in allWords )
{
    if ( words.Contains( word ) == false &&
        word.Length > 0 &&
        !IsJunk( word ) )
    {
        words.Add( word );
    }
} // end for each word in all words
```

The helper method takes each word it is passed, examines each character in the word, and returns false if it is ok (that is, it is not junk)

```
private bool IsJunk( string theWord )
{
    foreach ( char c in theWord.ToCharArray() )
    {
        if ( char.IsPunctuation( c ) ||
            char.IsDigit( c ) ||
            char.IsSymbol( c ) ||
            char.IsSeparator( c ) )
            return true;
    }
    return false;
}
```

Finally, we’ll end the event handler for the button by calling another helper method, *Display*, that will display the words in the two ScrollViewes by using the properties `Words` and `SortedWords`, to get the contents of the words collection in the desired order.

```
private void Display()
{
    Message.Text = words.Count + " unique words added. ";
    WordDisplay.Text = string.Empty;
    SortDisplay.Text = string.Empty;
    foreach ( string s in Words )
    {
        WordDisplay.Text += " " + s;
    }
    foreach ( string s2 in SortedWords )
    {
        SortDisplay.Text += " " + s2;
    }
}
```

That is the complete code for the Find page. If you want to make your program a bit more robust, you can disable (or make invisible) the buttons for the Search and Count pages until you

reach this point, because it is only now that you have the words collection needed to pass to those pages.

## The Search Page

Since this is a tutorial on page switching, I'll make short work of the other two pages, and as noted above, while I will show the usage of the Toolkit controls I won't, here, delve into their complexity. The Xaml for the Search page follows,

```
<UserControl x:Class="PageSwitching.Search"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:controls="clr-
namespace:Microsoft.Windows.Controls;assembly=Microsoft.Windows.Controls"

    Width="500" Height="185">
<Grid
    x:Name="LayoutRoot"
    Background="#FF000000" Height="Auto">
<Grid.RowDefinitions>
    <RowDefinition Height="0" />
    <RowDefinition Height="50*" />
    <RowDefinition Height="2*" />
    <RowDefinition Height="30*" />
    <RowDefinition Height="39*" />
    <RowDefinition Height="0" />
    <RowDefinition Height="80*" />
</Grid.RowDefinitions>
<Grid.ColumnDefinitions>
    <ColumnDefinition Width="1*" />
    <ColumnDefinition Width="4.6*" />
</Grid.ColumnDefinitions>
<TextBlock
    x:Name="wordPrompt" Text="The Word: "
    HorizontalAlignment="Right"
    Margin="0,12,5,0" Grid.Row="1"
    FontFamily="Verdana" FontSize="24"
    TextWrapping="Wrap" />
<controls:AutoCompleteBox
    x:Name="myAutoComplete"
    Margin="5,9,0,9" Grid.Column="1"
    Grid.RowSpan="1" Grid.Row="1"
    HorizontalAlignment="Left"
    Height="30" Width="210"
    FontFamily="Verdana"
    FontSize="14" />
<TextBlock
    x:Name="minPrefix"
    Text="Minimum Prefix Length:"
    Padding="5" FontFamily="Verdana"
    Margin="0,0,25,0" Grid.Row="3"
    HorizontalAlignment="Right"
```

```

        VerticalAlignment="Bottom"
        FontSize="18" />
<TextBlock
    x:Name="negOne"
    HorizontalAlignment="Left"
    VerticalAlignment="Bottom"
    Grid.Column="0" Grid.Row="4"
    FontFamily="Verdana"
    Text="-1" Margin="5,0,0,0"
    FontSize="14" />
<TextBlock x:Name="eight"
    Margin="0,0,5,0"
    HorizontalAlignment="Right"
    VerticalAlignment="Bottom"
    Grid.Column="0" Grid.Row="4"
    FontFamily="Verdana"
    Text="8" FontSize="14" />
<TextBlock x:Name="CurrentValue"
    Text="2" HorizontalAlignment="Right"
    VerticalAlignment="Bottom"
    Margin="0,0,3,5" Width="20"
    Grid.Column="0" Grid.Row="3"
    TextWrapping="Wrap"
    FontFamily="Verdana"
    Foreground="#FFF6300B"
    FontSize="18" />
<Slider x:Name="SetPrefixLength"
    Minimum="-1" Value="2"
    Maximum="8" SmallChange="1"
    LargeChange="2" Grid.Row="4"
    Grid.Column="0"
    Margin="24,0,20,0" />
<Border
    Height="Auto"
    x:Name="Border"
    HorizontalAlignment="Left"
    VerticalAlignment="Stretch"
    Width="500"
    Margin="0,0,0,0"
    Grid.Row="1" Grid.RowSpan="4"
    Canvas.ZIndex="-1"
    Background="#FF73B8F2" Grid.Column="0" Grid.ColumnSpan="2" />
<TextBlock Margin="0,0,0,0" Grid.Row="6"
    Text="" TextWrapping="Wrap" x:Name="TheWord"
    HorizontalAlignment="Center" VerticalAlignment="Center"
    FontFamily="Georgia" FontSize="48" Foreground="#FFFFFFF0"/>
<StackPanel HorizontalAlignment="Stretch" Margin="0,0,0,0"
    Width="Auto" Grid.Column="1" Grid.Row="6" x:Name="ButtonSP"
    Orientation="Horizontal">
    <StackPanel.Background>
        <LinearGradientBrush EndPoint="0.5,1" StartPoint="0.5,0">
            <GradientStop Color="#FF000000"/>
            <GradientStop Color="#FFF8527C" Offset="1"/>
        </LinearGradientBrush>
    </StackPanel.Background>
    <Button Height="40" x:Name="returnButton" Width="100"
        RenderTransformOrigin="0.5,0.5" Background="#FF00FF00"

```

```

        FontFamily="Georgia" FontSize="20" Content="Return"
        HorizontalAlignment="Left" VerticalAlignment="Bottom"
        Margin="5,0,0,0" Canvas.ZIndex="1"/>
    </StackPanel>
</Grid>
</UserControl>

```

The key thing to notice in the Xaml is the namespace (at the top of the file) used to include the toolkit code to enable the `AutoCompleteBox`:

```

xmlns:controls="clr-
namespace:Microsoft.Windows.Controls;assembly=Microsoft.Windows.Controls"

```

To make this work, you'll need to add a reference to the assembly `Microsoft.Windows.Controls` that came with the Controls toolkit.

## Search.xaml.cs

While the code to support the search page is a bit more complex than we saw in the previous example, the code for value passing is the same. We provide a private member variable of type `List<String>`,

```
private List<string> sortedWords = null;
```

and we populate that inside the implementation of `UtilizeState`,

```

public void UtilizeState( object state )
{
    if ( state != null )
    {
        sortedWords = state as List<string>;
        myAutoComplete.ItemsSource = sortedWords;
    }
}

```

Notice that we also set the `ItemsSource` property of the `AutoCompleteBox` to that list of strings. That “loads” the `AutoCompleteBox` with the words it will know how to suggest as the user types.

The page needs to be able to respond to a click on the return button and it also needs to be able to respond to a change in the slider that sets the minimum prefix length (the minimum number of letters the user must enter in order for the `AutoCompleteBox` to begin recommending words).

```

returnButton.Click += new RoutedEventHandler( returnButton_Click );
SetPrefixLength.ValueChanged +=
    new RoutedPropertyChangedEventHandler<double>(
SetPrefixLength_ValueChanged );
myAutoComplete.MinimumPrefixLength = 2;

```

```
myAutoComplete.LostFocus += new RoutedEventHandler( myAutoComplete_LostFocus
);
```

We have also created an event handler for the event that fires when the AutoCompleteBox loses focus so that we can take the chosen word and put it into the display in the lower left corner,



**Figure 8-9. Loss of Focus**

Here is the complete code from Search.xaml.cs,

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

namespace PageSwitching
{
    public partial class Search : UserControl, ISwitchable
    {
        private List<string> sortedWords = null;

        public Search()
        {
            InitializeComponent();
            Loaded += new RoutedEventHandler( Search_Loaded );
        }

        public void UtilizeState( object state )
        {
            if ( state != null )
            {
                sortedWords = state as List<string>;
                myAutoComplete.ItemsSource = sortedWords;
            }
        }

        void Search_Loaded( object sender, RoutedEventArgs e )
        {
            returnButton.Click +=
new RoutedEventHandler( returnButton_Click );
            returnCleanButton.Click +=
```

```

        new RoutedEventHandler( returnCleanButton_Click );
    SetPrefixLength.ValueChanged +=
        new RoutedPropertyChangedEventHandler<double>
( SetPrefixLength_ValueChanged );
    myAutoComplete.MinimumPrefixLength = 2;
    myAutoComplete.LostFocus += new RoutedEventHandler
( myAutoComplete_LostFocus );
}

void returnButton_Click( object sender, RoutedEventArgs e )
{
    Switcher.Switch( new Find() );
}

void myAutoComplete_LostFocus( object sender, RoutedEventArgs e )
{
    if ( myAutoComplete.Text != null && myAutoComplete.Text.Length > 1 )
    {
        TheWord.Text = myAutoComplete.Text;
    }
}

void SetPrefixLength_ValueChanged(
object sender,
RoutedPropertyChangedEventArgs<double> e )
{
    myAutoComplete.MinimumPrefixLength =
        (int) Math.Floor( SetPrefixLength.Value );
    CurrentValue.Text = myAutoComplete.MinimumPrefixLength.ToString();
}
}
}

```

## Count

The count page is not unlike the Search page. We begin by adding the two namespaces we need at the top of the Xaml page,

```

xmlns:controls="clr-
namespace:Microsoft.Windows.Controls;assembly=Microsoft.Windows.Controls"
xmlns:charting="clr-
namespace:Microsoft.Windows.Controls.DataVisualization.Charting;assembly=Micr
osoft.Windows.Controls.DataVisualization"

```

This requires that we add another reference, this time to the Microsoft.Windows.Controls.DataVisualization.dll that came with the toolkit.

The UI consists of a chart and a return button, and to support that, I've created two rows in my grid in the proportion of 6:1,

```

<Grid x:Name="LayoutRoot" Background="White">
  <Grid.RowDefinitions>
    <RowDefinition Height="6*" />
    <RowDefinition Height="1*" />
  </Grid.RowDefinitions>

```

Within the grid is a Chart object. I've named it letterFreqChart, the name to be used programmatically. The chart has three critical bindings:

- IndependentValueBinding
- DependentValueBinding
- ItemSource

In our case we'll set the first two in Xaml and the third programmatically, obtaining the ItemSource through the UtilizeState method.

## Measuring Frequency

One complicating factor here is that the chart wants to oppose the independent value (in our case the letter of the alphabet) against the dependent value (the number of words that begin with that letter). While we have a collection of words, we don't have that information, and we certainly don't have it in a single object. Thus we must pause briefly and create a new object that will provide the ItemSource for this chart.

```

public class Freq
{
  public int Count { get; set; }
  public char Letter { get; set; }

  public static List<Freq> Tally( List<string> words )
  {
    string prevChar = "a";
    int counter = 0;
    List<Freq> freqs = new List<Freq>();
    foreach ( string w in words )
    {
      if ( w.ToLower().StartsWith( prevChar.ToLower() ) )
        counter++;
      else
      {
        freqs.Add( new Freq(){ Letter=prevChar.ToLower()[0] ,Count=counter
});
        prevChar = w.Substring( 0, 1 );
        counter = 1;
      }
      // z
      freqs.Add( new Freq() { Letter = prevChar.ToLower()[0], Count = counter
} );
    }
  }
}

```

```

        return freqs;
    }
}

```

Frequency provides two properties, Count and Letter and one static method: Tally. You pass a List of strings to tally and it returns a List of Freq objects each of which contains a letter and the count of how many words in the original list began with that letter. Thus, simply, if you pass to Tally a list of these words: “also, author, away, both, bottom, change” you will get back a list of three Freq items, the first of which will have a property Count with the value 3 and Letter with the value ‘a’ and the second will have the property Count with the value 2 and Letter with the value ‘b’ and the final entry will have Count with the value 1 and Letter with the value ‘c’.

Here’s the Xaml for the Count page,

```

<UserControl x:Class="PageSwitching.Count"
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
xmlns:controls="clr-namespace:Microsoft.Windows.Controls;
assembly=Microsoft.Windows.Controls"
xmlns:charting="clr-
namespace:Microsoft.Windows.Controls.DataVisualization.Charting;
assembly=Microsoft.Windows.Controls.DataVisualization"
Width="400" Height="300">

<Grid x:Name="LayoutRoot" Background="White">
    <Grid.RowDefinitions>
        <RowDefinition Height="6*" />
        <RowDefinition Height="1*" />
    </Grid.RowDefinitions>

    <charting:Chart x:Name="letterFreqChart" >
        <charting:Chart.Series>
            <charting:ColumnSeries
                Title="Count"
                IndependentValueBinding="{Binding Letter}"
                DependentValueBinding="{Binding Count}" />
        </charting:Chart.Series>
    </charting:Chart>

    <Button x:Name="returnBtn" Content="Return"
        FontSize="18" Grid.Row="1" Height="30" Width="100"/>

</Grid>
</UserControl>

```

You see that the Independent Value is binding to the Letter property and the Dependent Value is binding to the Count property, obviously of a Freq object, through we’ve not yet supplied one; that will happen in the code. Here is the code from Count.xaml.cs

```

using System.Collections.Generic;
using System.Windows.Controls;

namespace PageSwitching

```

```

{
public partial class Count : UserControl, ISwitchable
{
    private List<Freq> freqs;
    private List<string> sortedWords;

    public Count()
    {
        InitializeComponent();
        Loaded += new System.Windows.RoutedEventHandler( Count_Loaded );
    }

    void Count_Loaded( object sender, System.Windows.RoutedEventArgs e )
    {
        returnBtn.Click += new System.Windows.RoutedEventHandler(
returnBtn_Click );
        Microsoft.Windows.Controls.DataVisualization.Charting.ColumnSeries cs =
letterFreqChart.Series[0] as
        Microsoft.Windows.Controls.DataVisualization.Charting.ColumnSeries;
        cs.ItemsSource = freqs;
    }

    void returnBtn_Click( object sender, System.Windows.RoutedEventArgs e )
    {
        Switcher.Switch( new PageSwitcher() );
    }

    public void UtilizeState( object state )
    {
        sortedWords = state as List<string>;
        freqs = Freq.Tally( sortedWords );
    }
}
}

```

We begin by declaring two member collections: a list of Freq objects and a list of strings. In the all important UtilzieState method sortedWords is filled from state, and then Tally is called, with sortedWords as the input, and the List of Freq objects as the returned value.

The second and third lines of the Loaded event handler assigns to the column series ItemSource the List<Freqs> as the data source,

```

Microsoft.Windows.Controls.DataVisualization.Charting.ColumnSeries cs =
letterFreqChart.Series[0] as
    Microsoft.Windows.Controls.DataVisualization.Charting.ColumnSeries;
cs.ItemsSource = freqs;

```

That second line is a bit ugly so let's unpack it,

```

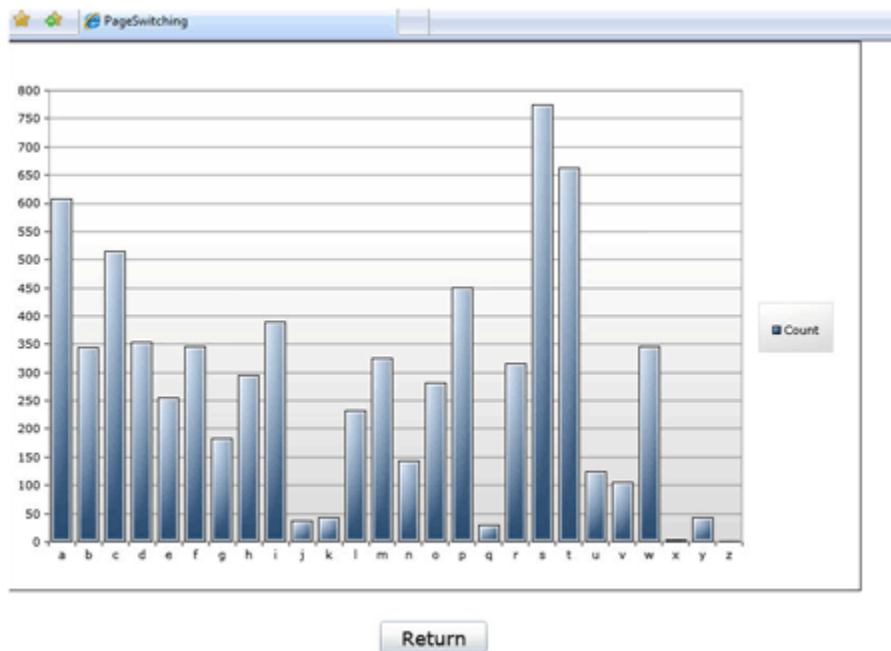
letterFreqChart.Series[0] // find the 1st element in the series for our chart

// cast it to be of type Column series
as Microsoft.Windows.Controls.DataVisualization.Charting.ColumnSeries

```

```
// assign it to a ref to an object of that type
Microsoft.Windows.Controls.DataVisualization.Charting.ColumnSeries cs =
```

Once we've made this assignment we have the Column Series and we can assign its ItemSource and the chart will display appropriately,



**Figure 8-10. Completed Chart**