

# Build WPF Screens Using Inheritance and Aggregation

If you have a WPF screen that is made up of two or more "sections," where each section has its own unique functionality, you might want to consider breaking each of those pieces of the screen into individual user controls and individual view model classes. This will help you build, run, and test each component. You can then aggregate the user controls into one control and inherit from one view model to the other to bring them all together. In this blog post, you are going to build upon the sample created in the post entitled "Basics of MVVM in WPF." Read and download that sample application to follow along with this blog post.

## The Sample Application

In the last blog post, you built a list of users using a ListView control and a view model class. The results look like the top of Figure 1. In this blog post, you are going to build a user control to display a label and text box for each field as shown in the bottom of Figure 1.

User ID	User Name	First Name	Last Name	Email
1	PShaffer	Paul	Shaffer	PShaffer@netinc.com
2	JSmith	John	Smith	JSmith@netinc.com
3	BJones	Bruce	Jones	BJones@netinc.com

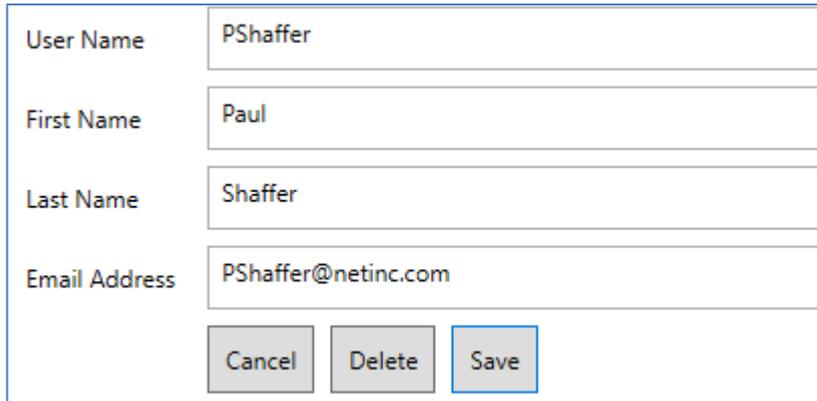
  

User Name	<input type="text" value="PShaffer"/>
First Name	<input type="text" value="Paul"/>
Last Name	<input type="text" value="Shaffer"/>
Email Address	<input type="text" value="PShaffer@netinc.com"/>

Figure 1: The sample application with a user list and detail user controls.

## Create User Detail User Control

Open the sample from the last blog post and add a new user control named **UserDetailControl.xaml** within the UserControls folder of the project. Figure 2 shows what this user control is going to look like.



User Name	PShaffer
First Name	Paul
Last Name	Shaffer
Email Address	PShaffer@netinc.com
<input type="button" value="Cancel"/> <input type="button" value="Delete"/> <input type="button" value="Save"/>	

Figure 2: The user detail screen to build.

Modify the `<Grid>` element so it has two columns and six rows as shown in the code below.

```
<Grid>
  <Grid.ColumnDefinitions>
    <ColumnDefinition Width="Auto" />
    <ColumnDefinition Width="*" />
  </Grid.ColumnDefinitions>
  <Grid.RowDefinitions>
    <RowDefinition Height="Auto" />
    <RowDefinition />
  </Grid.RowDefinitions>
</Grid>
```

After the closing `</Grid.RowDefinitions>` element, add the following label and text box controls.

```

<Label Grid.Row="0"
      Content="User Name" />
<TextBox Grid.Row="0"
         Grid.Column="1"
         Text="{Binding Path=Entity.UserName}" />
<Label Grid.Row="1"
      Content="First Name" />
<TextBox Grid.Row="1"
         Grid.Column="1"
         Text="{Binding Path=Entity.FirstName}" />
<Label Grid.Row="2"
      Content="Last Name" />
<TextBox Grid.Column="1"
         Grid.Row="2"
         Text="{Binding Path=Entity.LastName}" />
<Label Grid.Row="3"
      Content="Email Address" />
<TextBox Grid.Column="1"
         Grid.Row="3"
         Text="{Binding Path=Entity.EmailAddress}" />

```

After the labels and text box controls, add a stack panel for the three buttons.

```

<StackPanel Grid.Column="1"
            Grid.Row="4"
            Orientation="Horizontal">
  <Button Content="Cancel"
         IsCancel="True" />
  <Button Content="Delete"
         IsCancel="True"
         Click="DeleteButton_Click" />
  <Button Content="Save"
         IsDefault="True"
         Click="SaveButton_Click" />
</StackPanel>

```

Make sure to create the `DeleteButton_Click()` and `SaveButton_Click()` event procedures.

## Create User Detail View Model

In the `UserDetailControl` user control you see, you are binding to the properties within an Entity object. This Entity object is going to be in a view model you should create in the ViewModels folder named **UserDetailViewModel.cs**.

After creating this file, inherit from the `UserListViewModel` from the previous blog post. This provides you all the functionality of the `UserListViewModel` class, plus anything you add to the `UserDetailViewModel` class.

```
public class UserDetailsViewModel : UserListViewModel
{
}
```

Add the appropriate properties and methods to support the detail section of this screen. Add a property named *Entity* that is of the type **User**.

```
private User _Entity = new User();

public User Entity
{
    get { return _Entity; }
    set {
        _Entity = value;
        RaisePropertyChanged("Entity");
    }
}
```

## Override the LoadUsers Method

After loading the list of users, it would be nice to set the *Entity* property to the first item in the list. This will cause the binding on the user detail control to display the values for the user in the bound text box controls. Override the `LoadUsers()` method, call the `base.LoadUsers()` method, then check to ensure that the *Users* collection has some users. Set the *Entity* property to the first user in the *Users* collection.

```
public override void LoadUsers()
{
    // Load all users
    base.LoadUsers();

    // Set default user
    if (Users.Count > 0) {
        Entity = Users[0];
    }
}
```

## Add Save Method

Add a `Save()` method to the user detail view model. You are not going to write this code right now, but let's at least create the stub method that can be called from the click event of the Save button.

```
public void Save()
{
    // TODO: Save User
}
```

## Add Delete Method

Add a Delete() method to the user detail view model. You are not going to write this code right now, but let's at least create the stub method that can be called from the click event of the Delete button.

```
public void Delete()
{
    // TODO: Delete User
}
```

## Test User Detail Control

If you want, you can now test the UserDetailControl control to see if it displays the appropriate data from the view model. Open the UserDetailControl.xaml file and add an XML namespace.

```
xmlns:vm="clr-namespace:WPF.Sample.ViewModels"
```

Add a <UserControl.Resources> and create an instance of the UserDetailViewModel class.

```
<UserControl.Resources>
    <vm:UserDetailViewModel x:Key="viewModel" />
</UserControl.Resources>
```

Modify the <Grid> element to bind to the view model instance created the XAML.

```
<Grid DataContext="{StaticResource viewModel}">
```

Open the UserDetailControl.xaml.cs file and add a field to reference the UserDetailViewModel class.

```
UserDetailViewModel _viewModel = null;
```

Modify the constructor to retrieve the instance of the view model class created by XAML. Once you have the instance of the view model, invoke the `LoadUsers()` method to load the list and to set the *Entity* property to the first user.

```
public UserDetailControl()
{
    InitializeComponent();

    _viewModel = (UserDetailViewModel) this.Resources["viewModel"];

    _viewModel.LoadUsers();
}
```

Open the `MainWindow.xaml.cs` file and in the `MenuUsers_Click` event add an instance of the `UserDetailControl` control to the `contentArea.Children` collection.

```
private void MenuUsers_Click(object sender, RoutedEventArgs e)
{
    contentArea.Children.Add(new UserDetailControl());
}
```

## Try it Out

Run the application, click on the Users menu, and you should see the first user's detail information appear in the control you created.

## Back out the Changes

Now that you have tested the changes, remove the code you just added to the `UserDetailControl.xaml.cs` file and the changes you made to the XAML too.

# Aggregate the List and Detail Controls

You now have two stand-alone user controls that you need to combine into one. To accomplish this, aggregate the list and detail user controls onto another user control in order to build a screen that looks like Figure 1.

## Create User Maintenance User Control

Create a new user control named **UserMaintenanceControl.xaml** in the UserControls folder. Add a <ScrollView> within the <Grid>, and a <StackPanel> within the <ScrollView> as shown below.

```
<Grid>
  <ScrollView VerticalScrollBarVisibility="Auto">
    <StackPanel>

    </StackPanel>
  </ScrollView>
</Grid>
```

Build the solution to ensure all user controls are available in the toolbox. Open the Toolbox if it is not already visible. Drag and Drop the **UserListControl** control into the <StackPanel>. Drag and Drop the **UserDetailControl** control into the <StackPanel> below the **UserListControl** control. Your <StackPanel> control should look like the following.

```
<StackPanel>
  <local:UserListControl DataContext="{StaticResource viewModel}" />
  <local:UserDetailControl DataContext="{StaticResource viewModel}"
/>
</StackPanel>
```

**NOTE:** The prefixes in front of your user controls may be different than "local," if so, that is just fine.

## Create User Maintenance View Model

Just as you created a **UserDetailViewModel** that inherited from the **UserListViewModel**, you are going to add one more view model that inherits from the **UserDetailViewModel**. Add a new view model class named **UserMaintenanceViewModel.cs** in the **ViewModels** folder. You are creating this class to bind to the **UserMaintenanceControl** user control you just created.

```
public class UserMaintenanceViewModel : UserDetailViewModel
{
}
```

Open the **UserMaintenanceControl.xaml** file and add an XML namespace.

```
xmlns:vm="clr-namespace:WPF.Sample.ViewModels"
```

Add a `<UserControl.Resources>` element and create an instance of the `UserMaintenanceViewModel` class. Assign the key of "viewModel" to this view model object.

```
<UserControl.Resources>  
  <vm:UserMaintenanceViewModel x:Key="viewModel" />  
</UserControl.Resources>
```

Modify the `<Grid>` element to bind the `DataContext` to this resource.

```
<Grid DataContext="{StaticResource viewModel}">
```

By binding the view model instance to the `<Grid>`, all controls within the `<Grid>` are also given the same data context.

## Modify User List Control

The `UserListControl` user control is bound to the `UserListViewModel` through XAML. It is now time to remove that binding just like you did with the `UserDetail` user control. Open the **UserListControl.xaml** file and remove the `DataContext` attribute from the `<Grid>`.

```
<Grid DataContext="{StaticResource viewModel}">
```

Remove the `<UserControl.Resources>`.

```
<UserControl.Resources>  
<vm:UserListViewModel x:Key="viewModel" />  
</UserControl.Resources>
```

Remove the XML namespace that references the view model namespace.

```
xmlns:vm="clr-namespace:WPF.Sample.ViewModels"
```

Add the `SelectedItem` attribute to the `<ListView>` control to bind to the `Entity` property you added to the `UserDetailViewModel` class.

```
<ListView ItemsSource="{Binding Path=Users}"  
  SelectedItem="{Binding Path=Entity}">
```

Open **MainWindow.xaml.cs** and modify the `MenuUsers_Click()` event. Instead of adding an instance of the `UserListControl` user control to the `Children` collection of the `contentArea`, add an instance of the `UserMaintenance` user control.

```
private void MenuUsers_Click(object sender, RoutedEventArgs e)
{
    contentArea.Children.Add(new UserMaintenanceControl());
}
```

## Try it Out

Run the application.

Click on the Users menu and you should see the list of users appear.

The detail should be filled in with the information from the first user in the list.

Click on the other rows in the user list to see the detail information for each of them appear in the text box controls.

## Get a Reference to View Model

You will want to get a reference to the `UserMaintenanceViewModel` object in the code-behind of your detail user controls so you can invoke the `Save()` and `Delete()` methods. Open the **UserDetailControl.xaml** file and add the `Loaded` event.

```
<UserControl
  x:Class="WPF_MVVM_ListDetails.Samples.UserDetailControl"
  xmlns="..."
  xmlns:mc="..."
  mc:Ignorable="d"
  d:DesignHeight="450"
  d:DesignWidth="800"
  Loaded="UserControl_Loaded">
```

Open the **UserDetailControl.xaml.cs** file and add a using statement at the top of the file.

```
using WPF.Sample.ViewModels;
```

Create a private variable to reference the `UserMaintenanceViewModel` class.

```
// Create a reference to the view model
private UserMaintenanceViewModel _viewModel = null;
```

Modify the `UserControl_Loaded()` event to get a reference to the `UserMaintenanceViewModel` class. You retrieve this reference by casting the `DataContext` property on the `UserDetailControl` class to the `_viewModel` variable you created.

```
private void UserControl_Loaded(object sender, RoutedEventArgs e)
{
    // DataContext is instance of the UserMaintenanceViewModel class
    _viewModel = (UserMaintenanceViewModel)this.DataContext;
}
```

In the `btnSubmit_Click` event procedure, you may now invoke the `Save()` method on the `_viewModel` object as shown in the code below.

```
private void btnSubmit_Click(object sender, RoutedEventArgs e)
{
    // Save User
    _viewModel.Save();

    MessageBox.Show("User: " + _viewModel.Entity.UserName
        + " has been saved.");
}
```

You may also invoke the `Delete()` method on the view model class in the `btnDelete_Click()` event as shown below.

```
private void btnDelete_Click(object sender, RoutedEventArgs e)
{
    // Delete User
    _viewModel.Delete();

    MessageBox.Show("User: " + _viewModel.Entity.UserName
        + " has been deleted.");
}
```

## Try it Out

Run the application and click on the Users menu. Click on the Save button to see the message box appear. Click on the Delete button to see another message box appear.

## Other Examples of Aggregation

I have used this approach for many of my WPF applications. Figure 3 is a screen shot from the PDSA Developer Utilities ([www.pdsa.com/devutils](http://www.pdsa.com/devutils)). Each tab "Main," "Source Control," and "Log" is a different user control. Figure 4 shows a couple of tabs from the Computer Cleaner. Each of these tabs is very different from one another. Breaking each tab into their own separate user controls keeps the functionality separated so you don't have one very large WPF XAML file. This makes it easier to develop, test, and potentially reuse each user control.

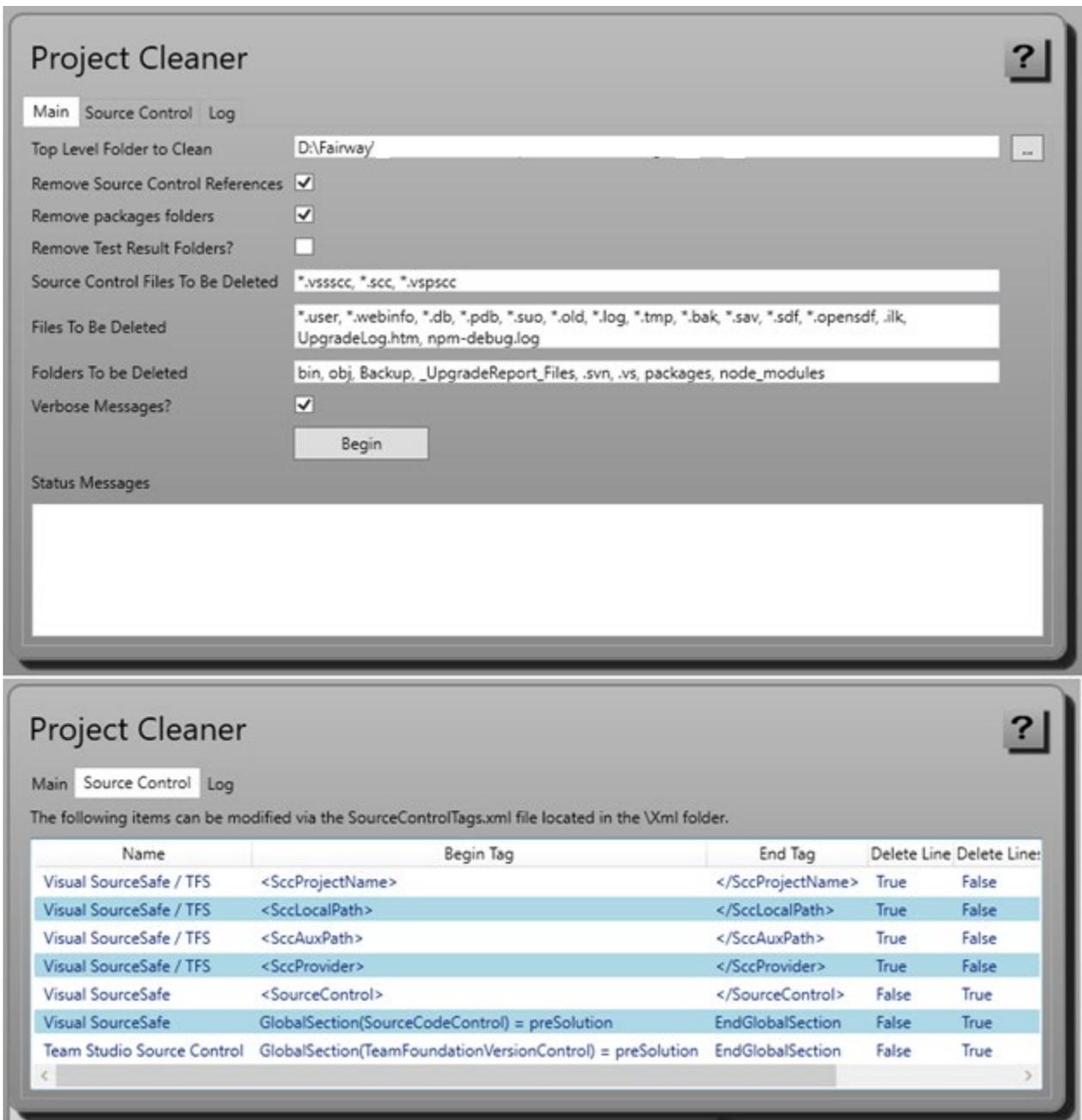


Figure 3: The Project Cleaner utility uses a separate user control for each tab.

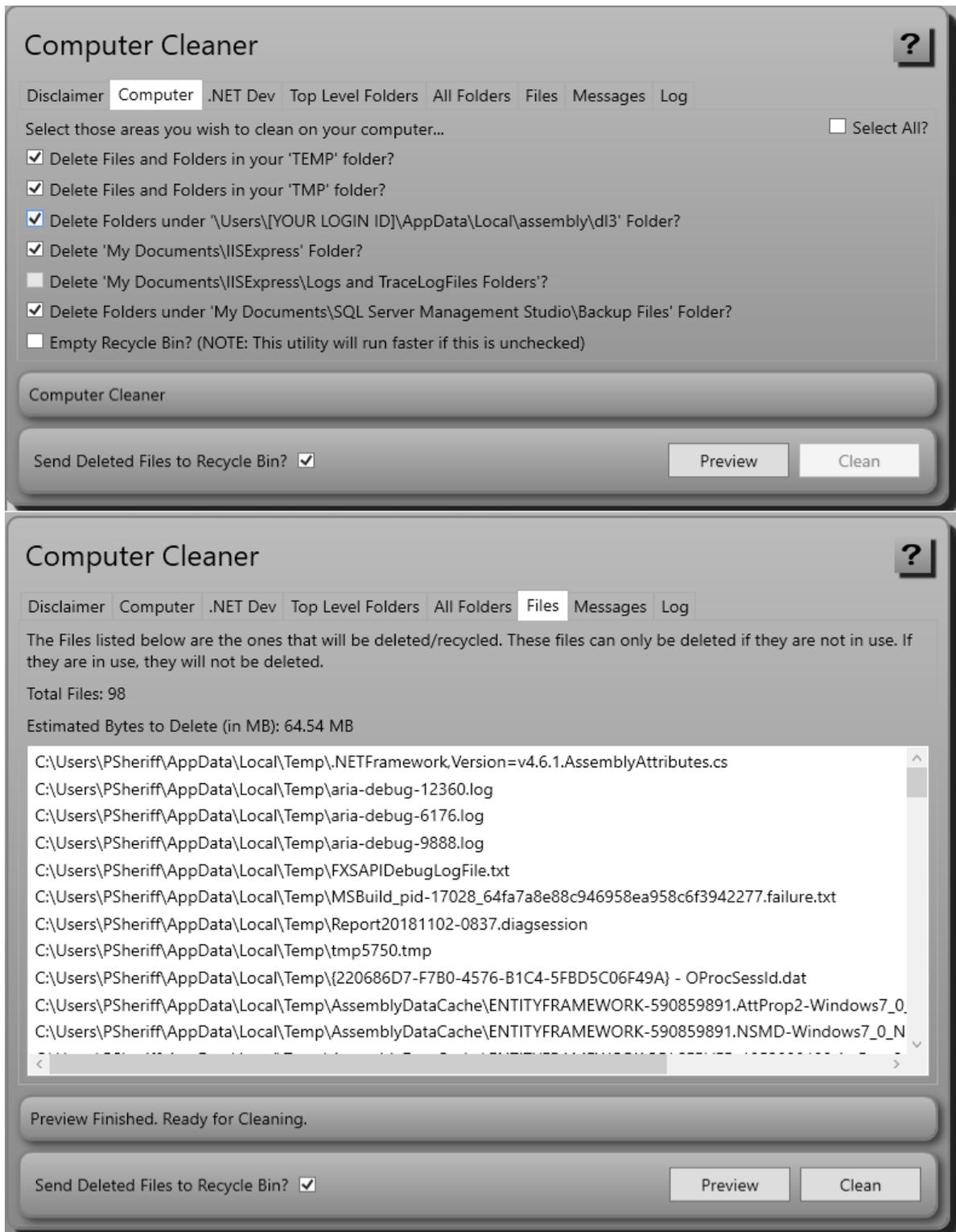


Figure 4: The Computer Cleaner utility uses a separate user control for each tab.

## Summary

In this blog post, you learned to use aggregation to build a screen from a couple of different user controls. Using aggregation allows you to build each piece of a screen separately. You may test the functionality separately, then using inheritance of the view models, you can bring all the functionality together. Sometimes, you must do a little hooking up of view models to test, then remove that functionality, but I find the extra work worth the effort.

## Source Code

NOTE: You can download the sample code for this article by visiting my website at <http://www.pdsa.com/downloads>. Select "Fairway/PDSA Blog," then select "Build Large WPF Screens Using Inheritance and Aggregation" from the dropdown list.