WPF Validation Scopes

A Bindable Project

Version: Working Draft

By Paul Stovell

Contents

[Background 3](#_Toc215247459)

[Introduction 3](#_Toc215247460)

[Getting Started 5](#_Toc215247461)

[Download 5](#_Toc215247462)

[Usage 5](#_Toc215247463)

[Concepts 7](#_Toc215247464)

[Validation Triggers 7](#_Toc215247465)

[Validation Providers 8](#_Toc215247466)

[Validation Display Strategies 8](#_Toc215247467)

[Validation Scope Nesting 9](#_Toc215247468)

[How To 10](#_Toc215247469)

[How To: Trigger validation when a user hits a button 10](#_Toc215247470)

[How To: Show a validation result on a control (Style) 11](#_Toc215247471)

[How To: Show a validation result on a control with an Adorner (Style + Template) 11](#_Toc215247472)

[How To: Show a validation result around multiple controls 11](#_Toc215247473)

[How To: Show extra information about an error 13](#_Toc215247474)

[How To: Show a list of validation messages 13](#_Toc215247475)

[How To: Show validation rollups on a tab control 13](#_Toc215247476)

[Known Issues 15](#_Toc215247477)

[Issue #1: Tab Controls with Validation Scopes don't validate until selected 15](#_Toc215247478)

[Issue #2: Virtualizing Panels 16](#_Toc215247479)

[Conclusion 17](#_Toc215247480)

[Attachments 17](#_Toc215247481)

[References 17](#_Toc215247482)

Overview

# Background

In 2006 I wrote an article for the Code Project entitled **Validation in Windows Presentation Foundation**. The article described the existing WPF Validation system, and towards the end demonstrated a simple control that solved some of the shortcomings in the system at the time.

As it turns out, I wasn't the only one affected by these shortcomings, and I have been continually surprised by how many people use the control that I posted. Unfortunately the control was not designed for performance or extensibility, and although I had revisited a few times over the years, I never published an update.

This article will introduce a new, revised control for performing advanced validation in WPF.

# Introduction

Out of the box, the WPF validation system provides many of the basic features needed in enterprise WPF applications. The out-of-the-box system provides support for three sources of validation:

1. ValidationRule objects applied to bindings at the UI level,
2. Validation performed by the data object itself through IDataErrorInfo,
3. Exceptions thrown in property setters

The system allows developers to provide templates and styles to control how validation messages are visualized. For many, this is all that is required from a validation system. However, some applications will have more complicated, richer requirements. The Validation Scope control described in the rest of this paper is designed to provide support for some of these more advanced validation scenarios.

I'll begin with a short "Getting Started" guide, and then explain some of the concepts behind the control, before finishing with a list of "How to" section for common needs. As a quick summary, the table below provides a feature matrix to show how the Validation Scope system differs:

|  |  |  |
| --- | --- | --- |
| Feature | Out of the box | Validation Scopes |
| Validate using IDataErrorInfo | Yes | Yes |
| Validate with inline rules on the UI | Yes | Not out of the box, but possible with provider model |
| Validate based on exceptions in property setters | Yes | No |
| Validate as soon as the data loads | Only for IDataErrorInfo, not Exceptions or ValidationRules | Yes |
| Validate only after a control has had focus | Only for ValidationRules and Exceptions, difficult with IDataErrorInfo | Yes |
| Validate only when explicitly triggered | No | Yes |
| Show a list of error messages | No | Yes |
| Auto-focus on an invalid control | No | Yes |
| Show errors around multiple controls at once (e.g., Password + RepeatPassword) | No | Yes |
| Show errors around the control irrespective of the Bindings (e.g., a panel around a TextBox, rather than the bound TextBox) | No | Yes |
| Richer error information (help topic links, localization ID's, categories, automatic corrections, etc.) | No | Yes |
| Pluggable validation providers (support for EntLib, or your own validation system) | No | Yes |
| Pluggable visualization strategies (Adorners out of the box, write your own) | No | Yes |

Validation Scopes in WPF

# Getting Started

This walkthrough will show you how to get started with the Validation Scope. It assumes some knowledge of IDataErrorInfo.

## Download

The latest code and releases can be found on CodePlex under the Bindable project:

* [www.codeplex.com/bindable](http://www.codeplex.com/bindable)

## Usage

Once you have a reference to the **Bindable.Windows.dll** assembly, you can use it from any VS 2008 project. The Validation Scope is designed to be used from XAML and wraps the set of controls you want validated.

To begin, from any WPF project add an XML namespace import for the Bindable Controls library.

<Window

...

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

>

Then, wrap your standard input controls with a ValidationScope:

<Bindable:ValidationScope

x:Name="ValidationScope"

ValidationSource="{Binding}"

ValidationTrigger="AutomaticAfterFocus"

>

<!-- TextBoxes and other standard WPF controls here -->

</Bindable.ValidationScope>

This assumes that the Window's DataContext is set to an object that implements IDataErrorInfo. The properties used above are:

* **ValidationSource** - the object being validated. Should be bound to an IDataErrorInfo object, or any other object that can be validated (see the section on Validation Providers later).
* **ValidationTrigger** - the trigger which causes validation. Options are:
  + **Automatic** (default) - validated as soon as the data loads
  + **AutomaticAfterFocus** - only validated once a control has been focused
  + **Manual** - only validated when told to explicitly, by a call to ValidationScope.Validate()
  + **AutomaticAfterManual** - behaves as per the Manual mode above, but then once validated, changes to Automatic behavior

Next, associate the input controls with the fields that they are interested in. This is done partly because it is faster than inspecting the bindings and also so that you can show validation information in controls other than the input control.

<TextBox

Text="{Binding Path=FirstName}"

Controls:ValidationScope.ValidateField="FirstName"

/>

Finally, you will need to write some styles to visualize the errors. Here is an example of a style for a TextBox:

<Style TargetType="TextBox">

<Setter Property="Background" Value="White" />

<Style.Triggers>

<Trigger Property="Controls:ValidationScope.IsValid" Value="False">

<Setter Property="ToolTip" Value="{Binding RelativeSource={RelativeSource Self}, Path=(Controls:ValidationScope.ElementValidationResult).Result.Message}" />

<Setter Property="Background" Value="Pink" />

</Trigger>

</Style.Triggers>

</Style>

You may wish to trigger validation at an explicit point in time, such as when a user clicks "Save". This can be done as follows:

private void SaveButton\_Clicked(object sender, RoutedEventArgs e)

{

if (\_validationScope.Validate().WasSuccessful)

{

// Perform save logic

}

else

{

\_validationScope.GetFirstValidationFailure().Element.Focus();

}

}

This is all that is required to hook the control up. Try experimenting with different ValidationTrigger values to see the effects they have.

# Concepts

This section details the concepts, designs and extensibility points behind the Validation Scope control.

The Validation Scope was designed primarily for extensibility. Where possible I have tried to make a few assumptions about how it will be used, but also to allow you to plug into the process. You can think of a Validation Scope as being the sum of the following parts:

|  |
| --- |
|  |
| **Figure 1: The Validation Scope is really just the sum of its parts.** |

## Validation Triggers

Validation Triggers tell the Validation Scope when to validate, a setting which is exposed by the ValidationTrigger property on the ValidationScope. They are an enumeration with four values:

* **Automatic** (default) - validated as soon as the data loads
* **AutomaticAfterFocus** - only validated once a control has been focused
* **Manual** - only validated when told to explicitly, by a call to ValidationScope.Validate()
* **AutomaticAfterManual** - behaves as per the Manual mode above, but then once validated, changes to Automatic behavior

When the trigger is specified, the Validation Scope internally creates and uses a ValidationStateManager to manage the knowledge inside each trigger. Although a custom implementation of the ValidationStateManager could be supplied, the triggers themselves are specified as an enumeration to make them more discoverable (they will appear in Intellisense and Blend). I could not think of any other trigger types, so I felt an enumeration is flexible enough, but if this is an area people would like to extend I will consider switching them to use a TypeConverter.

## Validation Providers

Validation Providers abstract the process of performing the validation and communicating the results back to the Validation Scope. They implement the IValidationProvider interface. The ValidationScope includes three default ValidationScopes:

* DataErrorInfoValidationProvider, for objects implementing IDataErrorInfo
* ValidatableValidationProvider, for objects implementing IValidatable, a custom approach to validation that supports richer error information.
* DetectableValidationProvider, which looks at the type of the object, and selects one of the two providers above.

This means that if an object implements IDataErrorInfo or another recognizable interface, the correct provider will be applied straight away. To supply a custom ValidationProvider, simply set the ValidationProvider property.

The IValidationProvider interface is simply:

public interface IValidationProvider

{

IEnumerable<IValidationFieldResult> Validate(ValidationContext context);

}

Where the ValidationContext provides information such as the fields to validate, the reason for validation, and so on.

## Validation Display Strategies

It is envisioned that validation results will be visualized in two main ways:

* Using styles that make use of attached dependency properties provided by the ValidationScope
* Using adorners, and the AdornedElementPlaceHolder

If this does not suit your needs, then as with providers, an interface IValidationDisplayStrategy is provided to allow you to hook into showing a validation failure. Simply implement the interface and assign it to the ValidationScope's ValidationDisplayStrategy property.

The interface for IValidationDisplayStrategy is:

public interface IValidationDisplayStrategy

{

void ShowValidationFailure(ValidationElementResult validationFailure);

void RemoveValidationFailure(ValidationElementResult validationFailure);

}

## Validation Scope Nesting

Validation Scopes can be nested as part of a hierarchy to provide validation rollups, as well as validation of child objects (for example, a root ValidationScope could validate an Order, while child ValidationScopes validate each OrderLineItem).

|  |
| --- |
|  |
| **Figure 2: How nested ValidationScopes may apply in a more complicated scenario.** |

When a ValidationScope is loaded, its parent ValidationScope is located through WPF's dependency property inheritance system. The relationship between Validation Scopes works as follows:

* When the parent ValidationScope is told to validate, it tells the children
* Child validation results are included in the results returned by the parent - that is, the parent ValidationScope will have a list of all errors including those by children
* In this sense, it can be said that the parent is "invalid" if any of the children are invalid, but not necessarily the other way around

As well as enabling the Validate() call to validate all children, this system also makes it possible to use validation "roll ups" (see the **How To** on that topic for more information). For example, the tab items in the screenshots below each display the number of errors displayed on each tab:

|  |
| --- |
|  |
| **Figure 3: Validation Rollups. Each tab shows the number of errors on the tab.** |

|  |
| --- |
|  |
| **Figure 4: The Friends tab, with a ListBox bound to the list of Friends exposed by the Student.** |

In this scenario, a ValidationScope would apply to the whole Window, with a nested ValidationScope applied to each TabItem. This would also allow each TabItem to show a list of errors for that tab, or for the root ValidationScope to show a list of all errors on the whole Window.

# How To

This section describes some common ways the Validation Scope can be used.

## How To: Trigger validation when a user hits a button

While it's nice to allow the UI to perform validation as the user interacts with the screen, there are times when you need to stop the user proceeding if there are errors. This can be performed by calling the Validate method whenever necessary.

This call returns a rich set of information about validation results, but the most common property to check would be the WasSuccessful property. For example:

private void SaveButton\_Clicked(object sender, RoutedEventArgs e)

{

if (\_validationScope.Validate().WasSuccessful)

{

// Perform save logic

}

else

{

\_validationScope.GetFirstValidationFailure().Element.Focus();

}

}

## How To: Show a validation result on a control (Style)

**TODO: This will show a style that uses the attached dependency properties to show error information.**

## How To: Show a validation result on a control with an Adorner (Style + Template)

**TODO: This will show the custom Validation Template and AdornedElementPlaceholder**

## How To: Show a validation result around multiple controls

Some validation information will concern more than one control. For instance, when two passwords do not match, instead of highlighting one password field or the other, it may be nicer to draw a box around the two invalid controls:

|  |
| --- |
|  |
| **Figure 4: The pink border is drawn around both of the controls when the passwords do not match.** |

This can be accomplished first by performing the validation in the data model. Here is how it might be done with IDataErrorInfo:

public string this[string columnName]

{

get

{

var result = new StringBuilder();

if (columnName == "Password" && (Password == null || Password.Trim().Length == 0))

{

result.AppendLine("Please enter a password.");

}

if (columnName == "PasswordsMatch" && (RepeatPassword != Password))

{

result.AppendLine("The passwords entered do not match.");

}

return result.ToString().TrimEnd();

}

}

The "PasswordsMatch" field referred to in the logic does not need to exist as a property (though it could).

From the markup, the following XAML would be sufficient to create the border:

<Border

Grid.Column="1"

Grid.Row="3"

Grid.RowSpan="2"

Style="{StaticResource **ValidationGroupBorder**}"

**Controls:ValidationScope.ValidateField="PasswordsMatch"**

/>

The following style would be necessary to make the border show/hide as the validation results change:

<Style TargetType="Border" x:Key="**ValidationGroupBorder**">

<Setter Property="Visibility" Value="Collapsed" />

<Setter Property="BorderThickness" Value="2" />

<Setter Property="BorderBrush" Value="Pink" />

<Setter Property="CornerRadius" Value="3" />

<Setter Property="Margin" Value="-1" />

<Style.Triggers>

<Trigger Property="**Controls:ValidationScope.IsValid**" Value="False">

<Setter Property="Visibility" Value="Visible" />

</Trigger>

</Style.Triggers>

</Style>

This approach can be used to show validation information in many other places, rather than directly on input controls. For example, you might use it to create a UI where each field has a label and space below it for the validation message instead (assuming you didn't want to use adorners to create the same effect).

## How To: Show extra information about an error

Validation results returned by the IValidationProvider simply have to implement the IValidationFieldResult interface, which just requires a message property and a few other simple properties. Each ValidationProvider has the responsibility of adapting its errors (the strings from IDataErrorInfo, for example) to this interface.

Using a custom ValidationProvider, or the out of the box ValidatableValidationProvider (**todo: I need to document this in more detail, but I'm not sure this document is the place**) your validation result adapters could contain much richer information. Examples are:

* A category, such as Error, Warning, etc.
* A message
* A localization ID
* A help topic link
* Auto correction abilities in some situations
* A video

The IValidateFieldResults are simply passed around the ValidationScope and available by the Result sub property on the attached ValidationElementResult property. In this way, your custom template would have easy access to whatever custom properties your ValidationProvider supplied.

## How To: Show a list of validation messages

**TODO: This can simply be done with a ListBox bound to the ValidationScopes's ValidationResult.ElementFailures property. The data template for the ListBox could show the error message, and even provide a hyperlink to select and focus on the invalid control. This would be great for jumping to items in a list.**

## How To: Show validation rollups on a tab control

Usually having controls highlight when they are invalid is enough to draw a user's attention, but with TabControls, this breaks down, since the errors may be hidden on a tab that the user cannot see.

One solution to this is of course to provide a list of messages in a very visible place. Another popular alternative, however, is to highlight that the tab has errors. Using some consistent color schemes should draw the user's attention.

|  |
| --- |
|  |
| **Figure 6: The number of failures shown at the top of the tab.** |

This can be accomplished by overriding the HeaderTemplate in the style for the TabItem. For example:

<Style TargetType="TabItem" x:Key="TabItemWithErrorRollup">

<Setter Property="HeaderTemplate">

<Setter.Value>

<DataTemplate>

<DockPanel>

<Border ClipToBounds="False"

IsEnabled="{Binding RelativeSource={RelativeSource   
 FindAncestor, AncestorType=TabItem},   
 Path=Content.ValidationResult.WasSuccessful}"

DockPanel.Dock="Right" Background="Pink"

BorderBrush="Red" BorderThickness="1"

Margin="10,-1,-6,-6" Width="20"

Padding="0" CornerRadius="2"

>

<TextBlock

Text="{Binding RelativeSource={RelativeSource FindAncestor,

AncestorType=TabItem},   
 Path=**Content.ValidationResult.ResultsCount.Current}"**

HorizontalAlignment="Center"

VerticalAlignment="Center"

/>

<Border.Style>

<Style TargetType="Border">

<Style.Triggers>

<Trigger Value="False" Property="IsEnabled">

<Setter Property="Visibility" Value="Visible" />

</Trigger>

<Trigger Value="True" Property="IsEnabled">

<Setter Property="Visibility" Value="Collapsed" />

</Trigger>

</Style.Triggers>

</Style>

</Border.Style>

</Border>

<TextBlock Text="{Binding }" />

</DockPanel>

</DataTemplate>

</Setter.Value>

</Setter>

</Style>

This style makes the assumption that the ValidationScope is the first child of the TabItem (by resolving the root TabItem, then binding to the Content property).

# Known Issues

There are two known issues that may cause problems when using the Validation Scope, which I have tried to outline and provide workarounds for below.

## Issue #1: Tab Controls with Validation Scopes don't validate until selected

A WPF TabControl contains a list of tabs, only one of which is ever "active" at any one time. Interestingly, WPF will load all of the children, but some children won't be loaded. This seems to occur mostly with both Content Controls and Items Controls.

A scenario to illustrate this is a TabControl, with the second TabItem holding an ItemsControl, with each item's DataTemplate using a ValidationScope. Interestingly, WPF will in fact load the ItemsControl, and the bindings will even be evaluated, but the child items in the ItemsControl won't be loaded until the tab is actually selected.

This seems to occur because the TabItem and its children are not Arranged until the tab is selected. It is during Arrange that the child items in the ItemsControl are created, and so it is at that point that the validation scope will be constructed.

A workaround, which may have unknown side effects, is to call both Arrange() and InvalidateArrange() on the tab item and any ItemsControls. This can be done using an attached dependency property provided with the Bindable.Windows assembly. First, include the namespace:

xmlns:Extensions="clr-namespace:Bindable.Windows.Extensions;assembly=Bindable.Windows"

Then apply the property:

<ItemsControl

...

Extensions:ItemsControlExtensions.CreateTemplatesOnLoad="True"

/>

I found that I only needed to apply this to the first control under the TabItem to get this to work, but sometimes it needed to be applied to all ItemsControls. This is very much an unsupported trick, but it seems to work. Apart from this approach, I know of no other ways to make TabControls, ItemsControls and nested ValidationScopes to work together.

## Issue #2: Virtualizing Panels

A related issue comes up when using any of the VirtualizingPanels in WPF.

These panels are usually used for controls such as Data Grids or the ListBox control, and increase application performance for large lists by only instantiating the templates for items that are currently visible. So, for example, if you bound to 100 items, templates for the first 10 might be instantiated and bound, and the rest would be done only as you scrolled.

As you might suspect, this means that if you had a nested ValidationScope in the DataTemplate of the item, it is not instantiated until it comes into view. This means that if the user called Validate on a parent scope, the children would not have even been instantiated yet, and so would not be able to participate in validation. This may to inconsistent results in the UI.

I can suggest three workarounds/solutions to this:

1. Don't use virtualization on large lists along with nested Validation Scopes. If possible, try reducing the number of items in the list so that virtualization is not required.
2. Alternatively, consider a UI design that does not require inline editing of every single item at once - perhaps selecting an item could turn it into "Edit Mode", at which time the template would switch to one with a ValidationScope, and then ensure it has been validated before switching back.

An alternative would be performing the validation in your bound root object as well. For instance, the Order could return a generic error, such as "one or more LineItems have errors".

Potentially, it could even tell the user each one, or you could use the ListBox's ItemContainerGenrator.GetItemForIndex method to locate and select the item in the list to bring their attention to it. However, this is not something the Validation Scope can easily accommodate out of the box.

Summary

# Summary

The Validation Scope control makes it easy to implement and support advanced validation scenarios that are not presently possible to do with the existing WPF validation system. These include features such as validation triggers, pluggable providers, error lists, rich error information and error rollups. Due to the closed nature of the WPF validation system, a radically different approach to validation was required in order to enable these features. The control also fixes many of the previous issues with my previous ErrorProvider control, especially in terms of performance.

I hope you find this new control useful and that it finds a home in your applications.

# Thanks

Special thanks go to Josh Smith for reviewing a prototype of the control, and to my coworker Jared Roughan for the many long chats about validation triggers. Thanks also to Gavin Rogers, Graeme Foster, Julien Tschäppät, and many others who provided feedback on the original control and ways to improve it.

Also thank you to you, for reviewing this draft document and the code, and making it better. Your feedback is most welcome.