Autonomous Validation: Extending IDataErrorInfo to Make Validation Simpler & More Robust

by R. Stacy Smyth

Introduction

One of the ways that IDataErrorInfo improves on the technique of writing custom validation classes is that it cuts down on the amount of custom XAML you need to write in order to enable validation. Still, the philosophy that IDataErrorInfo follows in automatically associating bindings with indexed error information could have been taken farther.

With IDataErrorInfo, it's easy to duplicate one snippet of code between your various validated bindings (ValidatesOnDataErrors ="True"), but you do need to make sure that that snippet is on one binding of every control that requires validation. Once you've done it a few times, it's usually easy to pick out which binding needs this snippet of code on each validated control, but you do still have to pick out the correct binding on every control. And if you want to validate something like a Label that doesn't usually have any bindings to the validated properties on the view model, you still wind up writing some XAML.

In an ideal world, we'd love to have our bindings automatically associated with the correct error information in IDataErrorInfo with no per-control XAML at all. Likewise, if we're validating Labels — which often aren't directly bound to the properties that determine their validation states — we'd like for them to deduce their validation states from the controls that serve as their targets, also with no XAML. In short, we wish the controls could just validate themselves, autonomously, without the view author having to think about which of their bindings require validation.

With such a system, view authors would mostly be able to forget that validation even exists, unless they were doing something unusual, like validating a Border. As long as the view model implemented IDataErrorInfo, and the controls got their data from the correct properties on the view model, validation would just happen.

Well, with IDataErrorInfo, we can't quite do it. But by extending IDataErrorInfo slightly with a new, derived interface — IAutonomousErrorInfo — we can.

This article and the accompanying sample application show how to build a validation system with these characteristics on top of the WPF validation extension points.

How It Works: An Overview

The fundamental approach used by IDataErrorInfo is a good one: storing validation results in an object that is indexed by view model property names to yield the error associated with each property. We're going to extend this interface in a derived interface named IAutonomousErrorInfo, but keep the underlying principle the same.

What we're going to change is that instead of the view author needing to know which control bindings should check this interface, the validation system will figure it out for us.

The key realization that makes this possible is that a mapping between the controls on the screen and the properties in the view model that should determine the controls' validation states is *already* implicitly defined by a combination of the bindings on the controls and a knowledge of which property of each control class is "the stuff entered by the user".

The first set of information — which properties of the controls are associated with what properties of the view model — is programmatically accessible by walking the logical tree of the view and looking at the bindings on each control.

The second set of information — which property of each control should have its binding source checked to determine the validity of the control — is largely provided by the class attributes of the controls themselves. If a control has a ContentProperty attribute defined, then 99% of the time, the source property on the view model that the content property is bound to is the property that we want to look up in IAutonomousErrorInfo to validate that control. If we're creating a control class ourselves and it isn't going to have a ContentProperty attribute, we can use a custom attribute (I named it the ValidationProperty attribute) to specify analogous information. For the very few control types defined by WPF where we might want to validate the control, but where no ContentProperty is given by WPF, we can hard-code the property to use for each class in our property-finder code. For example, if the control derives from Selector (ComboBox, ListBox, and TabControl), the relevant property is "SelectedValue". In the rare case where we want to override the usual property for an individual control, so that validation is based on some other property instead, we provide an attached property that can be specified in XAML for that control. (I called it AutonomousValidation.ValueToValidate).

Combining these two sets of information, we can create a system that takes advantage of this implicit association between controls and view model properties to perform validation autonomously. This system, which I'll call the "validation scanner", behaves in

the following way:

1. It's called whenever a change occurs to the set of errors associated with a view model (for example, the user just clicked "Save" and the validation logic — whether in the business layer or on the server — just returned results).
2. It walks the logical tree of the view, and for each control:
    a. Determines which property of the control should determine the control's validation state.
    b. Inspects the binding on that property, and thereby determines the name of the property on the view model with which that binding exchanges data.
    c. Checks the newly returned set of errors to determine whether any errors are associated with that property on the view model.
    d. Sets the validation state of the control accordingly.

How It Works: The Details

To avoid needing to add code to our views, the entire autonomous validation system (except for the IAutonomousErrorInfo interface on the view models) is implemented as an attached behavior.

Specifically, we set an attached property at the top of each view for which we'd like to enable autonomous validation:

```
xmlns:validation="clr-namespace:AutonomousValidation"
validation:AutonomousValidation.Errors="{Binding}"
```

This binds our attached property (AutonomousValidation.Errors), of type IAutonomousErrorInfo, to the view model that implements this interface. If you prefer, you can have a separate property of the view model implement IAutonomousErrorInfo instead, and bind to that. I chose to implement IAutonomousErrorInfo directly on the view model in the sample so that it would be more familiar to users of IDataErrorInfo.

The change handler for the Errors attached property adds a handler for the ErrorInfoChanged event of the IAutonomousErrorInfo. This gives us a handler that gets called whenever a change is made to the contents of the errors on the view model — which is just what we want for updating the validation state of the controls. This change handler kicks off a run of the validation scanner, described above.

One neat thing about this is that we don't need to add this attached property to any child views of the view on which we set the attached property — the sub-views are part of the logical tree of the parent view, so they'll be scanned and updated automatically.

Besides wiring up the attached behavior, the change handler for AutonomousValidation.Errors performs one other important function — it programmatically adds an additional attached property, the ValidationScanner property, to the view. Besides running a validation scan when the list of validation errors changes, the ValidationScanner stores a list of the properties that were invalidated by the previous scan, so that those properties can be restored to their valid state if the property names *don't* occur in the new error list in the IAutonomousErrorInfo.

When the validation scanner finds that a binding should be flagged as having invalid source data (remember, the WPF validation system is at bottom a system for invalidating bindings, not controls), it invalidates the binding by adding a ValidationRule to the binding. Specifically, it adds a validation rule of type ValidationFailureRule, which as you might guess always fails. If and when the problem with the source data is corrected, the validation scanner removes this validation rule and the binding becomes valid again.

One other thing the validation scanner does is handle the special case of Labels. With IDataErrorInfo, wiring up the validation of Labels required moderately clunky XAML on a per-Label basis. With the autonomous validation system, the validation state of a Label is determined by the validation state of the Label's target, which is generally what we want.

You may have noticed that the phrases "usually" or "almost always" crop up in this document with some frequency. That's because, fundamentally, the autonomous validation system reduces the amount of work required to implement validation in each view by having the validation scanner use class-level knowledge about which properties *probably* dictate the validation state of which controls. This means that, *generally*, you don't need to write any per-control XAML to enable validation. But any such system needs a way to provide for handling the unexpected. There are three ways to handle the special cases:

1. An attached property named AutonomousValidation.ValueToValidate is available so that you can, if necessary, specify in XAML that a control should be validated based on some property other than what would usually be assumed for that control class. (For classes like Border that don't *have* a usual property to validate, this is how you specify the property on the view model to check. By setting this

value to null, you can also tell the system not to validate the control at all.)

2. If you would like to use the standard WPF mechanisms for validating a particular binding — ValidatesOnDataErrors, ValidatesOnExceptions, or ValidationRule objects — just go ahead and use the standard WPF techniques. The validation scanner will automatically bypass any controls that are using these standard techniques.

3. In the rare case where the validity of a control is determined by some complex characteristic of the view model, and not by the validity of a single view model property (for example, too many check boxes have been checked), you'll need to create an additional property on the view model that can be bound to by `AutonomousValidation.ValueToValidate`. This additional property (I suggest a name ending in "Validity") does not ever need to have its value set, and does not need to support change notification — it merely needs to exist so that the IAutonomousErrorInfo can state that it has errors, which can then be passed to the control that cares about them.

As you can see, there are several different levels at which you can specify which property should be checked to determine the validity of a particular control. Here's the order of precedence:

1. Using old-style WPF validation on any binding of the control: ValidatesOnDataErrors, ValidatesOnExceptions, or ValidationRule objects.

2. Setting the AutonomousValidation.ValueToValidate attached property in the XAML of a control.

3. Specifying the ValidationPropertyAttribute on the control class.

4. Specifying the ContentPropertyAttribute on the control class.

5. Hard-coded exceptions for framework-provided control classes that do not possess a Content property.

The IAutonomousErrorInfo Interface

Here's the interface for your view models to implement in order to take advantage of autonomous validation:

```
public interface IAutonomousErrorInfo : IDataErrorInfo

    {

        /// <summary>

        /// Are there any errors?

        /// </summary>

        bool HasErrors { get; }


        /// <summary>

        /// Returns an IEnumerable of all of the view model property names that

        /// currently have errors associated with them.

        /// </summary>

        IEnumerable<string> PropertyNames { get; }


        /// <summary>

        /// An event that is raised whenever an error message is added or

        /// removed, and whenever the entire collection of errors is replaced.

        /// </summary>

        event EventHandler ErrorInfoChanged;

    }
```

That's it — two new properties and an event. For an example implementation, see BaseViewModelWithAutonomousErrorInfo.cs in the example program.

Variation: In fact, your view models don't *need* to implement this interface directly — if you'd like, you can implement this interface on a separate class (something like "ErrorInfo"), modify your view models to use a property of type ErrorInfo for storing their validation results, and have the AutonomousValidation.Errors property in your

view bind to the ErrorInfo object rather than binding to the entire view model.

The Example Program

The example program associated with this article provides a complete implementation of the autonomous validation architecture, as well as a simple view that uses this architecture to provide its validation. Buttons are provided to allow you to toggle the various controls between valid and invalid states.

The example view is defined in SubView.xaml and SubView.cs.

A sample implementation of IAutonomousErrorInfo is included in BaseViewModelWithAutonomousErrorInfo.cs. The appropriate implementation for your own app could be quite different, although I recommend looking at the sample first.

MainWindowModel.cs primarily contains command wiring to allow the user to validate and invalidate the controls in the example view.

Almost all of the meat of the autonomous validation architecture is in AutonomousValidation.cs and ValidationScanner.cs.