

Contents

- [Verification with a Silverlight UI](#)
- [Verification with a WPF UI](#)
 - [IdeaBlade.Core.INotifyDataErrorInfo](#)
 - [The INotifyDataErrorInfo workflow](#)
- [Verification with a WinForm UI](#)
 - [UI lockup](#)
 - [Unlock the UI with AutoValidate](#)
 - [Improving the user's experience](#)
 - [EnableAllowFocusChange and BeforeSet triggers](#)
 - [EnableAllowFocusChange and AfterSet triggers](#)
 - [Questionable user experience](#)
 - [Instance verification upon display](#)

Now that the application is detecting invalid data and throwing exceptions, we had better think about how we want to handle those exceptions and tell the user what is going on.

Verification with a Silverlight UI

System.ComponentModel.INotifyDataErrorInfo is a Silverlight-only interface that defines members that data entity classes can implement to provide custom, asynchronous validation support. Important Silverlight user interface controls including the *DataForm* and *DataGrid* check for the implementation of this interface by data entities to which they are bound and provide excellent messaging in the user interface to help end users learn about and correct validation errors.

Such Silverlight user interface controls do not depend upon exceptions being thrown in order to learn about validation problems. Instead, they subscribe to an *ErrorChanged* event published by the entity as part of its *INotifyDataErrorInfo* compliance. That event is raised by the data entity whenever it believes that a change to its state might be important to some control in the user interface. The UI control, learning of such a change, can then query the data entity via its *INotifyDataErrorInfo.GetErrors()* method to obtain a current set of validation errors, which it may then expose to the end user as its developer sees fit.

Because of the support for *INotifyDataErrorInfo* by Silverlight UI controls, most developers working in Silverlight will want to set the system default for *VerifierOptions.ErrorNotificationMode* to *Notify* rather than any setting that results in exceptions being thrown for validation errors.

Verification with a WPF UI

Currently there is no WPF equivalent of *INotifyDataErrorInfo*; and WPF user interface controls typically rely upon exceptions as their mechanism for learning about validation errors and stimulus to display information about them. Hence to accommodate WPF UI controls most developers will want to set the system default for *VerifierOptions.ErrorNotificationMode* to *ThrowExceptions* or *NotifyAndThrowExceptions*.

IdeaBlade.Core.INotifyDataErrorInfo

A *IdeaBlade.Core.ComponentModel.INotifyDataErrorInfo* interface has been implemented in DevForce 2010 with the same semantics as the similarly named class in the Silverlight CLR's version of *System.ComponentModel*. This interface is implemented by the DevForce *EntityWrapper*, from which all of your DevForce-generated *Entity* types derive. Therefore all DevForce-generated entities, Silverlight or otherwise, publish the *ErrorsChanged* event, maintain a *VerifierResultCollection*, and provide a *GetErrors* method that returns the collection of those. Because of the implementation of this interface by *Entity*, it is now possible for you to configure your entities to collect validation errors rather than throw exceptions. You may also do both, if you wish. Again, for most purposes, *VerifierOptions.ErrorNotificationMode* should be set in Silverlight applications to notify but not throw exceptions, and in WPF applications to throw exceptions (and, optionally, notify if the developer wishes to use that capability for other reasons than interaction with the commonly available UI controls).

A developer could choose, in a WPF application, to suppress exceptions and use the *INotifyDataErrorInfo* notification facilities instead. To do that, however, she would have to encode her UI explicitly so that it responds to the interface's *ErrorsChanged* event, as the commonly available WPF controls know nothing about any *INotifyDataErrorInfo* interface or behavior.

The *INotifyDataErrorInfo* workflow

Here is the validation workflow that properly occurs when the facilities of *INotifyDataErrorInfo* are in use (as is standard in Silverlight 4 applications):

1. The end user changes the value of a property.
2. Verifiers triggered by the changes to that property are fired.

- Each verifier that is run causes the *INotifyDataErrorInfo.ErrorsChanged* event to be raised. The event subscriber receives an [ComponentModel.DataErrorsChangedEventArgs](#) object that includes the name of the property whose error state may have changed. If there is no specific property whose error state may have changed, but yet the error state of the entity taken as a whole may have changed, then the subscriber receives a *null* value in [ComponentModel.DataErrorsChangedEventArgs.PropertyName](#).
- The control, which subscribes to the entity's *INotifyDataErrorInfo.ErrorsChanged* event (standard in *Silverlight* only!), receives notification that a change has occurred which may have affected the bound entity's error state. Said notification includes the name of the specific property whose error state has potentially changed, if there is a specific property. The control then calls *INotifyDataErrorInfo.GetErrors()* on the data entity, passing it the name of the property (if there was one) to get current information about the error state (of that property). *GetErrors()* returns a collection of *VerifierResults*. The UI then updates the displayed information about the error state of the property and/or the entity.

A developer wishing to make a WPF UI respond the same way would have to code all of the behaviors described in step 4 above.

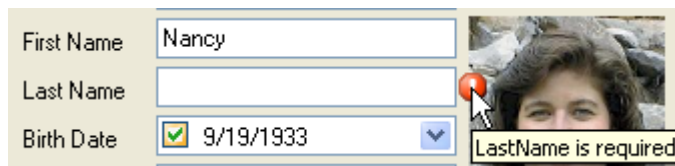
Verification with a WinForm UI

Data binding in WinForms is much more primitive than in either WPF or Silverlight. As a result, some of the processes described below are more complicated in this environment.

UI lockup

The UI is going to lock up the moment the user enters an invalid value into a verified UI control. That is any data entry control: *TextBox*, *DataPicker*, *ComboBox*, etc. The user will not be able to leave that control until she enters a value that passes validation – not even to close the form.

In this illustration, the user cleared the “Last Name”. The last name is required. The form displays an error bullet and prevents the user from moving out of the textbox.



How does the user recover?

If this were a grid, she could press the [Esc] key; it is “standard” for grid controls to restore the previous value when the user presses “escape.” How many users know that? In any case, this *TextBox* is not in a grid and pressing [Esc] does nothing but ring an annoying bell.

The user can press the standard key chord for “undo”: Ctrl+Z. How many users know that?

No, the most users will just keep entering new values until they find one that lets them out of the field.

Needless to say, a UI should apply the “lock up” enforcement technique sparingly. In the author’s opinion, it makes sense only for

- a value the user must know and is sure to know
- a value that must be correct immediately and at all times.

Dosage of a dangerous prescription drug would fit this bill. Few other properties qualify.

Unlock the UI with AutoValidate

Recall that the DevForce *Entity.BeforeSetValue* and *Entity.AfterSetValue* methods raise a *VerifierResultException* when the property fails validation. This exception bubbles up and out of the property setter.

Data binding traps the exception. During the data binding *Validate* event raised when the user attempts to leave the *TextBox*. and responds by locking up the form. Fortunately, WinForms .NET 2.0 makes it easy to change this response.

The key is the *System.Windows.Forms.UserControl.AutoValidate* property which takes one of the *System.Windows.Forms.AutoValidate* enumerations.

AutoValidate	Description
--------------	-------------

Inherit

Do what the parent UserControl does. The parent is the UserControl that contains this UserControl.

This is the default for new UserControl instances.

If there is no parent, the value is the default, EnablePreventFocusChange.

EnablePreventFocusChange

Prevents the user from leaving the control until the value passes validation.

EnableAllowFocusChange

Validate but permit the user to leave the control if validation fails.

Disable

Does not validate. Generally not a good choice.

Inherit is the default value for all new UserControls¹. Inherit means that the UserControl is governed by the AutoValidate setting of its parent UserControls, the UserControl that contains it.

The outer UserControl, typically a Form, doesn't have a parent so it is governed by the EnablePreventFocusChange setting.

If we never change the AutoValidate property on any UserControl, our application is governed by the setting in the Form which, as we have seen, is EnablePreventFocusChange, the setting that locks up the form. All UserControls within the Form are inheriting this behavior.

If we change the Form's AutoValidate property to EnableAllowFocusChange, the widgets on the Form will no longer lock up when the setter throws an exception. Neither will widgets on the contained UserControls because they inherit the parent Form's setting.

So the quick answer to UI lockup:

Change the Form's AutoValidate property to EnableAllowFocusChange

```
this.AutoValidate = System.Windows.Forms.AutoValidate.EnableAllowFocusChange;
```

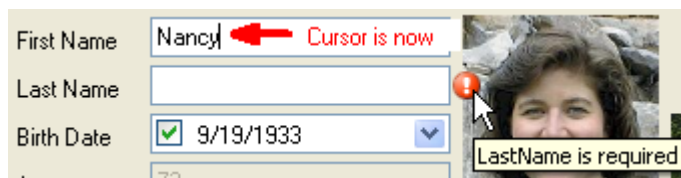
```
me.AutoValidate = System.Windows.Forms.AutoValidate.EnableAllowFocusChange
```

Improving the user's experience

EnableAllowFocusChange and BeforeSet triggers

AutoValidate.EnableAllowFocusChange works great for property verifiers governed by BeforeSet triggers.

The user can move out of the TextBox. Yet she can still see the error bullet protesting the lack of a "last name".



The TextBox remains cleared so we can see that there is a problem – or rather that there *was* a problem, that our intent to clear the name was invalid.

The LastName property itself was never actually changed. A BeforeSet trigger prevents the property setter from updating the object. At the moment there is a discrepancy between the business object property value and the corresponding widget control display property on screen².

We can see reveal the discrepancy and cure it by scrolling off of the "Nancy" employee and then returning to her. The TextBox refreshes with her current LastName property value which remains "Davolio".

EnableAllowFocusChange and AfterSet triggers

The behavior is different for verifiers evaluated in response to AfterSet triggers.

If we had a LastNameRequiredVerifier and set its ExecutionModes to InstanceAndOnAfterSetTriggers, the LastName property value would be empty, just as it appears in the TextBox. A AfterSet trigger causes validation *after* the property has been set with the "proposed value."

We can confirm this by scrolling off of the “Nancy” employee and then returning to her. The TextBox remains blank. The current LastName property value is empty.

However, we are no longer aware of the latent validation error. Our application does not validate the Employee upon display ... and that might be a user experience problem³.

At least it is not a data integrity problem – or doesn’t have to be. We must assume that the application follows our advice and ensures that every entity must survive “instance verification” before it can be saved. We further assume that the application has some mechanism to display errant entities and their problems. Perhaps a simple MessageBox will do.

This Employee will not survive validation, will not be saved, and the user will be told why.

Questionable user experience

This approach may be viable if little time can pass between data entry and instance verification.

Some applications attempt a save whenever the user moves off the current screen. The user will never lose sight of the LastName error bullet and the save effort will reveal all latent problems with this employee.

Many applications delay save and allow the user to move around among entities with pending changes. That’s how our tutorial works. Users can make a change to “Nancy”, scroll to “Andrew” and make changes to him, then scroll back to “Nancy” to continue her updates.

In this kind of workflow, the user may not remember that there is a problem with the “Nancy” object for minutes or hours. When the application finally tells the user about this problem, the mental context is long gone and the application will be perceived to be “unfriendly”.

There is another, potentially greater risk. The user may make a critical business decision base upon what is visible on the screen. That data could be in error. The user won’t know it if she scrolled off and then back on to the record.

If this risk is serious, the application must behave differently whenever the UI displays a new object – a new Employee in our example.

Instance verification upon display

One approach would be to perform instance verification whenever the currently displayed object is changed.

1. [^] UserControl is the base class for developer designed screens. System.Windows.Form inherits from UserControl. Individual “UI widgets” such as TextBox do not inherit from UserControl.
2. [^]
3. [^] We could write code to perform “instance validation” whenever the Employee changed. We could capture the VerifierResults and display them as well as light up bullets next to each widget. The code is not hard to write but it’s not utterly trivial either. We’ll describe an approach that achieves something of that effect using a different technique.