Build your own CAB #15 – MicroControllers

Posted by Jeremy Miller on October 22, 2007

After a bit of a hiatus and a fair amount of pestering, I'm back and ready to continue the "Build you own CAB" series. The point isn't really to go build a drop in replacement for the Composite Application Block (CAB), but rather to understand the design patterns you can use to build maintainable WinForms applications. I'm hopeful that the material in this series will be useful for both users of the CAB and those who are rolling application specific architectures sans CAB. The previous posts have been mostly written at the perspective of a single screen. For this post I want to go down to the granular control level before coming back up to finish the series with content on coordinating multiple screens and composite applications. My longer term goal is use the BYO CAB series as the kernel of a book on design patterns for fat clients and composite applications, so I'd love any kind of feedback no matter how negative.

MicroControllers

I introduced what I call the MicroController pattern in my last installment. The basic idea is to use a small controller class to govern an atomic part of the screen, usually all the way down to the individual control or a small cluster of controls. The assumption is that it'll be easy to reuse the very small controller classes across screens and also to test these little critters under an xUnit microscope. I've found that it's often advantageous to then aggregate those MicroController's to create powerful behaviors. I'm going to take this idea a little bit farther and show some usages of the MicroController pattern to create an alternative to Data Binding for screen synchronization. When I first wrote about this idea in My Crackpot WinForms Idea, I said that I'd do a further writeup if the technique was successful. I've used it long enough now to know that I'm happy with the results overall, and more importantly, I'd like some feedback on this approach before I think about using it again. As I'll try to demonstrate in this post, I'm claiming the MicroController approach leads to improvements in productivity and testability.

Background

I'm going to refer to two different projects in this post.

1. *TradeCapture 1*: A project that I did late last year and early this year. We used WinForms data binding and ran into major stumbling blocks with automated testing
2. *TradeCapture 2*: One of the projects I've been working on since April. I'd used the MicroController strategy on previous projects, but this is the project where all the techniques shown here were developed and tested. The design of *TradeCapture 2* was heavily influenced by my interpretation of our struggles from *TradeCapture 1*.

I'm using slightly obfuscated examples from *TradeCapture 2*. An early, but functional version of the MicroController strategy is in the StoryTeller codebase at http://storyteller.tigris.org/svn/storyteller/trunk/src/StoryTeller.DataBinding. If you down load the entire code tree at http://storyteller.tigris.org/svn/storyteller/trunk you'll find a couple small screens that use the techniques from this article. ReSharper "Find Usages" is your friend.

Upcoming posts...

How I'm using StoryTeller to tes
FubuMVC
Building a "Lookup" html conver
FubuMVC
FubuMVC's Configuration Mode
Sauce"
Managing Script dependencies
FubuMVC
Authorization and FubuMVC
Continuations
Composing Views with FubuMV
Extensible Model Binding with F
Introducing "Bottles"
Modular Packaging with FubuM
Self-Installing Apps w/ FubuMV(
Routing and Behavioral Conven
FubuMVC
What Should I Learn?

Blogroll

Follow me on Twitter
FubuMVC on GitHub
My GitHub Page
StoryTeller on GitHub
StructureMap on GitHub

Archives

January 2012
September 2011
July 2011
June 2011
May 2011
April 2011
March 2011
January 2011
December 2010
September 2010
August 2010
July 2010
June 2010
May 2010
March 2010
February 2010
January 2010
December 2009

What's the Problem?

If you stop and think about it, there's a huge amount of repetitive tasks that you need to do at the individual control level. Here's a partial list of the tedious chores that can easily fill up your day on a WinForms project:

- Bind controls to a property of the Model
- Fill dropdown boxes with reference data fetched from some sort of backend
- Use the ErrorProvider to display validation errors by control
- Disable or hide controls based on user permissions
- Capture change events off controls to trigger some other sort of behavior
- Do "IsDirty" checks
- Reset all the values back to the original state
- Inside of an automated test, you need to be able to find a certain control and then manipulate it
- Format data in screen elements. In my project we need to make some textbox's accept numeric values in the form "1k" or "1m" to enter large numbers. In other applications you may have other formatting and parsing rules repeated over controls.

The designer time support in Visual Studio makes it easy to create these types of behavior from scratch. That's great, but it leads to a lot of duplication, code noise, and difficulty in making behavioral changes across multiple screens harder. In the last bullet point in my list above, I had to go back and accept "1b" as the value 1,000,000,000 in textbox's representing monetary amounts. It wasn't any big deal because I only had one single method in a single MicroController class to modify. But what if I'd created that behavior through implementing event handlers separately for each textbox? That'd be a lot of code to duplicate and change.

Screen Synchronization with MicroControllers

For a variety of reasons I wanted more predictable screen synchronization than Data Binding provides. Add in the drag (I'll talk about this at length in the next BYO CAB post) of Data Binding in automated testing scenarios and I was ready to try something different. I ended up using a scheme that I'd previously applied to Javascript heavy DHTML clients. I would create a MicroController class for each type of Control that knew how to bidirectionally synchronize data from the Model classes to the Controls. The mechanics for each type of control (textbox, radio buttons, checkbox, listbox, etc.) are slightly different, but the goals and intentions are basically the same. As an example, for every type of control I might want to:

- Synchronize the value of a property on the Model with a Control on the screen
- Apply changes from a Control value back to a property on the screen
- Attach error messages to a Control by the name of the property the Control is bound to

- Simulate a “Click” event
- Register for changes in the Control’s value
- Reset the Control values back to the original property value of the Model class
- Determine if the Control is “dirty”
- Enable or disable Controls

The key is to make each type of Control/MicroController look the same for basic operations. I do this by making each MicroController implement the IScreenElement interface partially shown below:

```
public interface IScreenElement
{
    string LabelText { get; }
    string FieldName { get; }
    Label Label { get; set; }
    ErrorProvider Provider {set;}
    …
    void Bind(object target);
    bool ApplyChanges();
    void SetError(string errorMesssage);
    void Reset();
    string GetError();
    void RegisterChangeHandler(VoidHandler handler);
    void Update();
    void SetErrors(string[] errorMessages);
    …
    void Disable();
    void Enable();
    void Click();
    event VoidHandler OnDirty;
    void StopBinding();
}
```

At the moment, I’ll focus on getting information between a Control and a single property of a Model class (say we have a textbox called “nameTextbox” that is bound to the “Name” property of a Person Model class). The bidirectional binding of the property data to the control is done in the two methods in bold. Calling Bind(object) will take the value from the Model object property designated by the FieldName property and make that the value of the underlying Control. Likewise, calling ApplyChanges() will take the value of the underlying Control and push the value back into the Model object. The workflow of screen binding is similar enough to pull most of the functionality into a ScreenElement superclass. Here’s the implementation of the Bind(object) method from ScreenElement.

```
public virtual void Bind(object target)
{
    try
    {
        _target = target;
        _originalValue = (U) Property.GetValue(target, null);
        updateControl(_originalValue);
    }
    catch (Exception e)
    {
        string message = string.Format(“Unable to bind property ” + Property.Name);
        throw new ApplicationException(message, e);
    }
}

public void updateControl(U newValue)
{
    // Set the latch while we’re setting the initial value of the
    // bound control
    _latched = true;
    resetControl(newValue);
    _latched = false;
}

protected abstract void resetControl(U originalValue);
```

Let’s say that we do have a textbox bound to the “Name” property of a Person class. In the Bind(object) I keep a reference to the Person object that I’m binding to, then I use reflection to get the “Name” value off of the Person object, then I call updateControl() to actually set the Text property of the textbox. To build a specific ScreenElement for a textbox I just had to override the resetControl() template method in the TextboxElement class.

September 2005
August 2005
July 2005
June 2005
May 2005
April 2005

What others have said…

Duncan on What an Amazing (C You’ve Discovered
paweł paul on FubuMVC Learns Spark
paweł paul on I’m looking for so testers for StoryTeller (OSS tool testing)
paweł paul on Serenity
paweł paul on My Programming

```
        // BoundControl is a Textbox
        protected override void resetControl(object originalValue)
        {
            BoundControl.Text = originalValue == null ? string.Empty : _format(originalValue);
        }
```

There's really not too much to it. "_format" is a reference to a delegate that can be swapped out at configuration time to handle differences like the number of decimal points in numeric fields. Just for completeness sake, here's the same method in the PicklistElement (the implementation of ScreenElement for comboboxes)

```
        private IPicklist _list = new NulloPicklist();

        protected override void resetControl(object originalValue)
        {
            _list.SetValue(BoundControl, originalValue);
        }
```

Aggregating MicroControllers

By themselves, a single MicroController isn't all that usefull, but aggregating them together is a different story. I use a class called ScreenBinder to perform aggregate operations across a collection of IScreenElement MicroControllers. The public interface for IScreenBinder is down below:

```
    public interface IScreenBinder : IScreenElementDriver
    {
        void UpdateBinding();
        void FillList(string fieldName, IPicklist list);
        object BoundObject { get;}

        …

        event VoidHandler OnChange;
        void BindScreen(object target);
        bool ApplyChanges();
        void ShowErrorMessages(Notification notification);
        void ClearErrors();
        void StopBinding();
    }
```

As I said before, ScreenBinder keeps an internal ArrayList<IScreenElement> member. When you add an IScreenElement to ScreenBinder it also adds a reference to the proper ErrorProvider for validation error display and sets up event listening for change events. Since ScreenBinder aggregates change events for all of its IScreenElement children, you can simply listen for a single event on IScreenBinder for enabling "Submit" and "Cancel" type buttons when any element changes.

```
        public void AddElement(IScreenElement element)
        {
            // Attach the ErrorProvider to the new IScreenElement
            element.Provider = _provider;
            _elements.Add(element);

            // Go ahead and Bind the new IScreenElement
            if (isBound())
            {
                element.Bind(_target);
            }

            // Register for all OnDirty events
            element.OnDirty += fireChanged;
        }
```

Now that we have a collection of IScreenElement children, we can bind the whole collection to the Model object one child at a time.

```
public void BindScreenTo(T target)
{
    withinLatch(delegate
                    {
                        foreach (IScreenElement element in _elements)
                        {
                            element.Bind(target);
                        }
                        _target = target;
                    });
}

private void withinLatch(VoidHandler handler)
{
    _isLatched = true;
    handler();
    _isLatched = false;
}
```

Notice the lines of code in bold. When the ScreenBinder is binding each IScreenElement to the Model object it sets its internal "_latch" field to true. That's important because we don't want "IsDirty" event notifications popping up in the middle of the initial data binding. The "latch" strategy comes into play in the method fireChanged() that raises the ScreenBinder's OnChange event. That was a huge source of heartburn to me and my team on *TradeCapture 1*. One of my core goals for the MicroController strategy on *TradeCapture 2* was to systematically control the event latching in the screen synchronization.

```
public event VoidHandler OnChange;
private void fireChanged()
{
    if (_isLatched)
    {
        return;
    }

    if (OnChange != null)
    {
        OnChange();
    }
}
```

The more predictable data binding by itself was a win, but I had other design goals as well. The remainder of the post is a rundown of those goals and how I used MicroControllers to achieve these goals.

Goal: Make the View behavioral code easier to maintain and verify by inspection

Like I said earlier, the design time property editor is great for writing small behavioral and formatting functionality from scratch, but can you look at a screen in the designer and "see" what's wired up to what? I'll answer that one with "you can't." To find out what the screen is doing, which events are wired and to what, and what screen elements are bound to which property you've got to click on all of the elements and scroll through the Properties tab in Visual Studio. The information you need to maintain or patch the screen is scattered all over the place. I got very frustrated on *TradeCapture 1* with how hard it was to understand the behavior of complex screens.

To combat that problem in *TradeCapture 2* I wanted the wiring of the View to be compressed into a readable form without

sacrificing the straightforward speed of using the designer. Setting up all the MicroController objects was going to lead to ugly, verbose code that made the readability of the code worse. What I ended up with is a Fluent Interface to configure MicroControllers against all the screen elements in a readable format. To remove a little more friction, I wrote a crude codegen tool that simply spits out a class with constants for all of the public properties for my Model classes like this one below for my Trade class.

```
public class TradeFields
{
   public static readonly string Description = “Description”
   public static readonly string Status = “Status”
   public static readonly string TradeId = “TradeId”
   …
}
```

Hey, if you have to work in a statically typed language you might as well take advantage of it right? Having the property names in Intellisense is definitely better than strings every which way.

Now that I have the property names as constants I can configure a screens behavior with code like this snippet down below:

```
        public void Bind(IScreenBinder binder)
        {
            _binder = binder;

            _binder.BindProperty(TradeFields.Trader).To(traderCombobox).WithLabel(traderLabel)
                .FillWith(ListType.Trader);

            _binder.BindProperty(TradeFields.Strategy).To(strategyCombobox).WithLabel(strategyl
                .FillWith(ListType.Strategy);

            // LegalEntity & Book
            _binder.BindProperty(TradeFields.Book).To(bookCombobox)
                .FillWith(ListType.Book)
                .WithLabel(bookLabel)
                .RebindOnChange()
                .OnChange(delegate {_presenter.BookChanged();});

            _binder.BindProperty(TradeFields.LegalEntity).To(legalEntityCombobox).WithLabel(leg

            _binder.BindProperty(TradeFields.CounterParty).To(counterPartyCombobox).WithLabel(c
                .FillWith(ListType.Counterparty);

            _binder.BindProperty(TradeFields.TradeDate).To(tradeDatePicker).WithLabel(tradeDate
                .OnChange(delegate { _tradePresenter.TradeDateChanged(); });

        }
```

In this one method I’m completely defining both the data binding from control to property and the wiring of the View to its Presenter for OnChange events (look at the two snippets of code above in bold). I’m arguing that this type of data binding and declarative attachment of behaviors and event tagging is better for maintainability than classic designer driven Data Binding because all of the relevant functionality is boiled down into such a smaller area of the code. I can scan this code and understand what the screen is doing and spot errors in the screen wiring. The speed issue of the initial write is at least on par with Data Binding through the designer by simply enabling Intellisense.

I’m not sure I really got all the way to my goal of clean, expressive language to describe the desired screen behavior because of the inherent limitations of C# 2.0. My thinking is that this approach will shine much more in IronRuby or even in C# 3.0. I thought a little bit about rewriting this entire data binding scheme as an open source project, but I think I’m shelving that idea indefinitely. I do think it would be an interesting project for IronRuby and WPF someday.

Goal: Reuse minor screen behavior

Since coming to New York I've worked on two different "Trade Capture" applications. I've observed a lot of sameness of programming tasks across the screens in the two projects and I think I learned a lot of lessons from the first that have made the second more successful. One of the things we hit in the first project was this scenario: you have some sort of property that's set by choosing a tab or a radio button in the screen. In *TradeCapture 1* we wrote manual code that set the property on the underlying Trade object anytime the tab selection was changed. It's subtle, but I'd almost call that an intrusion of business logic into the presentation code. Even if you make the case that that code was definitely presentation related, it added repetitive code and noise to the View. Much worse is the fact that that behavior wasn't covered by an automated test because we gave up early on automating tests through the screen.

In *TradeCapture 2* I saw that same scenario coming and built a MicroController for that repetitive behavior. In this case I have several instances of an enumeration property, with a series of radio buttons representing each possible enumeration value. In the code we need to know how to set the property anytime a radio button is selected, and also to activate the correct radio button when an existing Trade is viewed.

I wrote two classes, a MicroController for each radio button called RadioElement and a class to manage the radio button group called RadioButtonGroup (If you're curious, the source code is at the links for each). The actual code for each class isn't that interesting, but I think this code is:

```
// FXOptionTradeFields is just a codegen'd class with a bunch of constants for
// the field names of the Model class
binder.BindProperty(FXOptionTradeFields.ExerciseType).ToRadioGroup<ExerciseType
    .RadioButton(americanOptionButton).IsBoundTo(ExerciseTypeEnum.American)
    .RadioButton(europeanOptionButton).IsBoundTo(ExerciseTypeEnum.European)
    .RadioButton(bermudanOptionButton).IsBoundTo(ExerciseTypeEnum.Bermuda);
```

All this code does is setup a RadioButtonGroup and a series of RadioElement objects to govern the data binding of the three radio buttons bolded above. It's just a little bit of Fluent Interface to make the attachment of the behavior be as declarative as possible.

Now, back to the idea of testability. In the next post in this series I'll show how to use the MicroController strategy to test through the screen, but for now let's assume that we're forgoing automated tests on the screen itself. In that case we've still made a gain. The behavior of the radio button binding is now more or less declarative, increasing the solubility of the code to make this code easier to trouble shoot by mere inspection.

You are reusing this code across different radio button groups, so it would help if the reused code were tested pretty thoroughly. While testing at the application or screen level is difficult, simply unit testing the MicroController classes in isolation was pretty simple. It's easy to forget, but the Control classes in WinForms are just classes. You can instantiate them on the fly in code at will. Here's what the unit tests for RadioElement look like:

```
[TestFixture]
public class RadioElementTester
{
    private EnumTarget _target;
    private RadioButton _button;
    private RadioElement<FakeEnum> _element;

    [SetUp]
    public void SetUp()
    {
        _target = new EnumTarget();
        _target.State = FakeEnum.TX;
        PropertyInfo property = typeof(EnumTarget).GetProperty("State");

        _button = new RadioButton();
        _element = new RadioElement<FakeEnum>(property, _button, new RadioButtonGroup<FakeE
    }

    [Test]
    public void CheckTheRadioButtonIfTheEnumerationValueMatches()
    {
        _element.BoundValue = FakeEnum.TX;
        _element.Bind(_target);
        Assert.IsTrue(_button.Checked);
```

```
        }

        [Test]
        public void DontCheckTheRadioButtonIfTheEnumerationValueDoesNotMatch()
        {
            _target.State = FakeEnum.MO;
            _element.Bind(_target);
            Assert.IsFalse(_button.Checked);
        }

        [Test]
        public void ApplyChangesWhenTheButtonIsChecked()
        {
            _target.State = FakeEnum.MO;
            _element.Bind(_target);
            _button.Checked = true;

            // After applying the changes, the RadioButton was selected,
            // so the _target.State property should have been overwritten
            // with the value of the RadioElement.BoundValue
            _element.ApplyChanges();
            Assert.AreEqual(_element.BoundValue, _target.State);
        }
    }
```

Goal: Eliminate noise code in the Presenters

My first exposure to Model View Presenter architectures was an application built along Passive View lines. Passive View does a lot to promote testability, but you can end up with massive Presenter classes. The Presenter in a Passive View screen can become a dumping ground for all of the screen responsibilities if you're not careful. One of my goals with my current screen architecture was to move to a Supervising Controller architecture and reduce the "noise" code in my Presenter's by pushing tedious tasks back into the View. For example, I've written code to grab a list of values from some sort of backend service and stuff it into a setter on the View to fill dropdown boxes more than enough times in my life. That kind of stuff just ends up bloating the Presenter with trivialities. I want the "signal" in the Presenter to be screen behavior, not a bunch of boilerplate code filling in dropdown lists and attaching validation messages.

What does it really take to fill a dropdown list with values? My end goal was to simply say in the code that I want ComboBox A to be filled with List B without having to worry about the mechanics every time. I'm going to do this a hundred times or better over the life of the project, so it might as well be easy. I came up with a syntax that looks like this:

```
            _binder.BindProperty(TradeFields.Strategy).To(strategyField).WithLabel(strategyLabe
                .FillWith(ListType.Strategy);
```

The screen that contains this code has a ComboBox named "strategyField." The code in bold above is directing the MicroController for "strategyField" to fetch the list of "Strategy" values to fill the ComboBox. To pull this off I need two things. The first thing is a way to describe a dropdown list to allow for variances in type or display/value members. I use a class called Picklist for this that has the public interface shown below:

```
    public interface IPicklist
    {
        void Fill(ComboBox ComboBox);
        void Fill(ListBox listBox);
        …
    }
```

The second thing I need is a single reference point to access IPicklist objects by key. To that end I use an interface called IListRepository:

```
    public interface IListRepository
    {
        IPicklist GetPickList(ListType type);
```

```
    }
```

The actual concrete implementation is gathering up list data from a couple different sources, but it's all accessible in a common way by calling the GetPickList(ListType) method. All the MicroController has to do now is grab an instance of IListRepository, find the right IPicklist, then fill its ComboBox control. Here's the method in PicklistElement that does just that:

```
public void FillWithList(ListType listType)
{
    // Grab IListRepository from StructureMap.  It's actually a singleton,
    // but we don't have to care about that here
    IListRepository repository = ObjectFactory.GetInstance<IListRepository>();
    IPicklist list = repository.GetPickList(listType);
    FillWithList(list);
}
```

The example code here is very specific to my application, but I've used the basic idea of a "ListRepository" and "Picklist" on a couple different UI-intensive projects to great effect. You could happily roll your own equivalent.

Goal: Enable Model-centric validation via the Notification Pattern

In an earlier installment on model centric validation, I made the case for putting validation logic into the Model class where that logic is easier to test and share across screens. I also introduced the Notification pattern as a way to transmit the validation messages tagged by field, but I purposely put off the mechanics of displaying those validation messages on the screen. So, here's the situation, I'm in the View now and I've got a Notification object with lots of validation errors, how do I get those associated to the correct control? Elementary my dear Mr. Watson. All we need to do is have ScreenBinder loop through its IScreenElement children. Inside of ScreenBinder is this method called ShowErrorMessages(Notification) that:

1. Loop through each IScreenElement
2. Query the Notification object for all of the error messages for the IScreenElement.FieldName
3. Tell the IScreenElement to display these error messages. If the error messages are blank, the IScreenElement will clear out all error display.

```
public void ShowErrorMessages(Notification notification)
{
    // Hack to replace "FieldName" with "Label Text" inside of validation messages
    foreach (IScreenElement element in _elements)
    {
        notification.AliasFieldInMessages(element.FieldName, element.LabelText);
    }

    // Call each ScreenElement to display the error messages for it's field
    // It's important to loop through each element so that the element knows
```

```
            // to show no messages if it's field is valid
            foreach (IScreenElement element in _elements)
            {
                string[] messages = notification.GetMessages(element.FieldName);
                element.SetErrors(messages);
            }

            // I need to refactor the propagation of the children into a Composite pattern
            // here so that child IScreenBinder's just look like IScreenElement.  Someday soon
            foreach (KeyValuePair<string, IScreenBinder> pair in _children)
            {
                Notification childNotification = notification.GetChild(pair.Key);
                pair.Value.ShowErrorMessages(childNotification);
            }
        }
```

At least for now, each individual IScreenElement is given a reference to the screen's ErrorProvider object. Since the IScreenElement has a reference to both the ErrorProvider and the Control, setting the error messages becomes pretty simple:

```
        public void SetErrors(string[] errorMessages)
        {
            SetError(string.Join(MESSAGE_SEPARATOR.ToString(), errorMessages));
        }
        public void SetError(string errorMesssage)
        {
            // _provider is the ErrorProvider for the screen
            if (_provider == null)
            {
                return;
            }
            _provider.SetError(_control, errorMesssage);
        }
```

I should point out here that it is perfectly possible to use the BindingDataSource from WinForms 2.0 to effect a similar effect to make a connection from field to control. All the same though, I like my way better.

Goal: Control extraneous "I just changed" events

Here's a common screen scenario: the submit and undo buttons should only be enabled when something on the screen changes. Easy enough, you just listen for "onchange" type events on all of the elements that you care about. There's one little potential problem though. Those onchange events can and will fire during the initial loading of the control data. Or when the list items of a ComboBox changes. Or some sort of formatting is applied to a textbox. At these times you want to disregard the onchange events. You can simply unregister the change events while the original binding operation is taking place or use a "latch" to stop the propagation of events when it's not appropriate. I tend toward using the "latch" approach.

As you've probably guessed, we can bake the "latch" idea into a MicroController to put the onchange event latch closer to each Control. I register event handlers against a MicroController instead of directly against the bound controls.

```
            binder.BindProperty(FXOptionTradeFields.CurrencyPair)
                .To(currencyPairField)
                .FillWith(ListType.CurrencyPair)
                .RebindOnChange()
                .OnChange(delegate { presenter.CurrencyPairSelected(); })
                .WithLabel(currencyLabel);
```

Intercepting the onchange events in the MicroController first enables the ability to “latch” the propagation of “onchange” events during the act of binding the Model to the control. Most of my MicroControllers inherit from a class called ScreenElement. In the fragment of code from ScreenElement below the updateControl() method is responsible for setting the value of the inner Control. Before ScreenElement calls through to the resetControl() method to actually set the value of the control it sets a field called _latch to true and sets _latch back to false as soon as resetControl() returns. Note the code in bold below.

```
public virtual void Bind(object target)
{
    try
    {
        _target = target;
        _originalValue = (U) Property.GetValue(target, null);
        updateControl(_originalValue);
    }
    catch (Exception e)
    {
        string message = string.Format("Unable to bind property " + Property.Name);
        throw new ApplicationException(message, e);
    }
}

public void updateControl(U newValue)
{
    // Set the latch while we're setting the initial value of the
    // bound control
    _latched = true;
    resetControl(newValue);
    _latched = false;
}
```

The call to resetControl() will fire off an event because the value of the Control is changing, but that event will not be propagated on if the latch is set. Look at the code in bold below:

```
protected void elementValueChanged()
{
    // Guard clause.  Don't do anything if we're latched
    if (_latched)
    {
        return;
    }

    ApplyChanges();

    // Re-calculate validation errors for only this field
    SetErrors(Validator.ValidateField(_target, _property.Name));

    // Call the other registered handler's for the onchange
    // event for this control
    foreach (VoidHandler handler in _handlers)
    {
        handler();
    }

    if (isDirty())
    {
        fireDirty();
    }
}
```

Each subclass of ScreenElement needs to capture the appropriate “onchange” event and call the elementValueChanged() method inherited from the base. Here’s the implementation from CheckboxElement:

```
public class CheckboxElement : ScreenElement<CheckBox, bool>, IToggleable
{
    public CheckboxElement(PropertyInfo property, CheckBox checkBox) : base(property, chec
    {
        checkBox.CheckedChanged += checkBox_CheckedChanged;
    }

    protected override void tearDown()
    {
        BoundControl.CheckedChanged -= checkBox_CheckedChanged;
```

```
        }
        void checkBox_CheckedChanged(object sender, System.EventArgs e)
        {
            elementValueChanged();
        }

        …
    }
```

Catch Jeremy's next "Build Your own CAB" adventure in Yes, Virginia. You CAN test the user interface. Coming to an RSS aggregator near you in about four of Jeremy's train ride commutes.

About Jeremy Miller

Jeremy is the Chief Software Architect at Dovetail Software, the coolest ISV in Austin. Jeremy began his IT career writing "Shadow IT" applications to automate his engineering documentation, then wandered into software development because it looked like more fun. Jeremy is the author of the open source StructureMap tool for Dependency Injection with .Net, StoryTeller for supercharged acceptance testing in .Net, and one of the principal developers behind FubuMVC. Jeremy's thoughts on all things software can be found at The Shade Tree Developer at http://codebetter.com/jeremymiller.
View all posts by Jeremy Miller →

This entry was posted in Build your own CAB, Design Patterns. Bookmark the permalink. Follow any comments here with the RSS feed for this post.

← It's Smart to Challenge the ASP.NET Status Quo

How are you building WinForms applications? What's hard about it? →