1 Thoughts, Rants and Arguments: My DevTeach 2007 Rollup

In case you're wondering why the Shade Tree Developer has been content free the last couple of weeks, I've been preparing for the talks that I gave at DevTeach this week. DevTeach 2007 is now in the books, so I'll trade in this strange Spoken Word tool for the more familiar Written Word and the wonderful magic of CTRL-Z. In other words, it's time to do one big ol' stream of consciousness post on everything I was thinking about this past week. I will surely get flamed for a bit of this, so stay tuned.

Enough rambling, here's what was trickling through my head after last week:

Hanging with the Cool Guys

These kinds of conferences are great mostly for the people you meet and interact with, and DevTeach was extraordinary in this regard. One of the things I miss about Austin is the constant entertainment that Bellware instantly provides in any situation.

I think I could easily spend all day talking to Jean-Paul. You should definitely take in his week long .Net training courses if you ever get the chance.

For months now, I've been looking forward to meeting the Israeli contingent, and we've even accused Bellware of helping stage the Agile track just to manufacture an opportunity to bring them to North America. Roy, Udi, and Oren (Ayende) are three of my favorite authors and blogger's. Meeting them in person was every bit as cool as I thought it was going to be. I love Ayende/Oren's passion and energy for development. Udi took some time with me to demonstrate his approach to DDD within an SOA. The dude's seriously got his head on straight. I got to take part in great conversations every single day on all things software related, but I think the most memorable episode at DevTeach was talking with Roy Osherove about our experiences being new fathers.

I was on the Microsoft & OSS panel moderated by Ted Neward. That's going to be one for the scrapbook (even if we didn't really get very controversial until the P&P stuff came up).

I also got to hang around a little bit with Rob Daigneau. Cool guy.

Seeing how the other half lives

I worry a little bit sometimes about living in an Agile echo chamber. I bet better than half of the RSS feeds I subscribe to are from Agile practitioners. I'm pretty sure that the majority of the people who subscribe to my blog either individually or through the main CodeBetter feed are already pro-Agile to some degree or another. I used to purposely read Hacknot (dropped because he doesn't write much and he's just too curmudgeonly) and Software Reality (dropped because they're just plain dumb) just to make sure I got a more balanced diet of intellectual content.

During the last day of DevTeach I purposely went to a talk in the architecture track about building application frameworks. I knew the talk was going to be far outside the realm of Agile thought, but I love talking software design so it sounded like an interesting talk. I just wanted to see how the non-Agile guys were approaching design and software construction and see if there is something else I need to be paying attention to.

From the first slide through the very end I wanted to jump out of my chair and argue with him on almost every single thing he put forward. The nameless architect (as I will refer to him throughout the rest of the post) lost me immediately. Let's see if I can remember all the things that bugged me (the sarcastic language is obviously mine). Build the application framework upfront. Divide the team into castes. The upper caste builds the application framework because they're more suited for the application design. The lower caste just can't be trusted, so we'll strait jacket them with the application framework to protect them from themselves. Liberally sprinkle your framework with all the hook methods you think you'll need later.

Honestly, I think the code he was showing as a sample had some design flaws. The code munged a couple concerns into one piece of code (always a big no, no in my book). He was using hand rolled factory methods and inheritance everywhere. I challenged him on that asking why he wasn't favoring composition and using an IoC tool to wire it together. He didn't particularly give me much of a response. I think I could have eliminated a whole bunch of his infrastructure code by merely using an IoC tool, and gotten a heckuva lot more flexibility out of the framework to boot. More importantly, the code wasn't testable — and you know for me that testability is an almost mandatory prerequisite for being "good" code. The maxim "favor composition over inheritance" is probably older than I am, and it's borne out by my experience. I'm a little shocked to see an experienced architect so blithely ignoring this advice.

The end result is that this nameless architect and I could not possibly coexist on the same project. My longstanding opinion that traditional architects are useless if not downright harmful remains completely intact. I guess you wouldn't be slightly surprised to find out this guy is a big proponent of software factories as well. Of course, this guy had more people in his talks and better evaluations than me, so what do I know?

I'm always hearing that us Agile guys can learn stuff from the older, non-Agile software traditions. You always have something to learn from more experienced people (note that I did NOT say older), but that particular traditional architect could really stand to spend some time listening to what we Agilists have to say about design and how we work. The nameless architect was presenting what I would consider to be the best of 90's era software design, but we've learned plenty since then. The absolute worst part of that talk was looking around and seeing essentially no overlap in faces between the folks coming into the Agile track and the people I saw in the architecture track. I was horrified at some of the design advice this guy was spitting out, but the other attendees were rapt. We've got a problem there.

Software Factories – The CASE Tools Fiasco 2.0?

I'm not old enough to know the details, but I've heard plenty of stories about how people invested in CASE tools during the 80's and early 90's only to see the movement bomb. I do think there are some incremental gains to be had from Software Factories (I'm thinking of things like Jay Flower's CI Factory and the software factory-like project and code generation in Ruby on Rails). The problem is that I also think that Software Factories look a heckuva lot like BDUF wrapped up into new, spiffier clothes. My resistance mostly comes down to three things that I think are going to happen:

People are going to freeze in Analysis Paralysis designing their own software factories outside of any real development work much the same way people waste so much time designing speculative frameworks today People are going to follow the software factory output no matter what. Again, this is BDUF at its worst. Design is never done. The Software Factory “guidance” and generation may work in most cases for your app, but I promise that there will be more than enough edge cases where the software factory pattern fails. You better be awake. People are once again going to waste time creating an environment where bad or mediocre developers can create acceptable software instead of investing in people.

Any older guys out there want to comment on the CASE tools from the 80’s? I couldn’t find any information on it at all, but I’ve heard plenty of horror stories.

By the way, there *IS* a Silver Bullet

After the first day I started to count how many conversations boiled down to an issue of safety versus power. It came up talking to a Sql Server guru who wanted to do data access through sproc’s just to protect the database against the developers. It came up several times in discussions about using Aspect Oriented Programming and Metaprogramming. The nameless architect was trying to design a framework that forced developers into a strait jacket to prevent them from doing something wrong with the obvious implication that the developers just couldn’t be trusted.

It comes down to a choice of:

Enabling developers with the maximum in language and technical power and trusting them to do the right thing Making development “safe” for lesser skilled developers by taking choices away from developers

I read recently an argument that choice #2 does far more to hamper the efforts of your best developers than it does to make weaker developers more productive. I agree. So wrapping this whole stream of dialogue up, I’ll say that there is a silver bullet(s), it’s:

Skill. Knowledge. Experience. Passion. Discipline.

In other words, it’s the people stupid! We need to invest much more in our people and the general skill level of the people doing the work instead of wasting so much energy on newer and newer forms of governance. We pay a very high price in productivity by not doing a better job of creating strong developers.

Why can’t you trust your developers?

Since this came up soooo many times, I have to ask — Why can’t you trust your developers to do the right thing? If you don’t trust the developers, what can you do about it? The hiring process is an obvious first step, but you’ve largely got the people that you’ve got. Can you work with them? Communicate better? And how do you know that the fault isn’t with you? What is it that made you trustworthy? Can it be repeated for another person? Do you really think you’re that special?

Or is the general population of developers so bad that this is hopeless and we really do need to worry about better and better ways to wring usefulness out of bad developers?

I hope this isn't true, but it seems to be the prevailing opinion of the majority of people I've spoken to in the last year or so.

There are no smart guys, there's only us

When I write blog posts giving any type of advice I'm always waiting for some smarter, wiser developer to set me aside and say, "let a real expert handle this." One of the other Agile track speakers said to me that he just doesn't feel like he's that great, and he's always waiting for someone to call him out on that. A great developer I worked with in the past had a phrase for this feeling: "There are no smart guys, there's only us." There is no expert on the hill. No one out there knows everything and has all the answers. Look, I'm an elitist with a sizable ego, but I know there are plenty of things I don't know all that well and there are plenty of folks that intimidate me. You know what though — they're just guys.

Not naming any names, but most of you know who I'm talking about, I joined an elite consultancy four years ago thinking that I'd get a chance to learn at the feet of the masters. I definitely learned a tremendous amount there, but I walked in as a technical lead from day one. Maybe that sounds strange, but that was a huge disappointment for me.

So what am I saying here? I guess I'm saying that

No one's infallible and all knowing Don't sell yourself short compared to other people

To take one last shot at Software Factories, if there are no smart guys, why the heck should I trust some guy who doesn't know me or my app to do my design for me? I certainly don't trust that nameless architect to design software, so why the fuck should I take a Software Factory from this guy?

XP environments are so much more fun

I sat in on the lunch time session on Pair Programming from a couple folks from Oxygen Media. Pairing is good, blah, blah, blah. What really struck me about their talk was how much I miss working in a dedicated Extreme Programming shop. I think XP teams and XP environments have far more energy and collaboration than traditional teams. You can just feel it in the room. My client environment is pure "cubesville" and the place is just flat out dead — excuse me, professional. Work is work, but all things being equal I'd rather have an environment that allows me to have fun and be effective. Then again, being effective is fun to me even without having toys everywhere.

I've met 5-6 of the Oxygen Media team now in various events in NYC, and I gotta say that I think it's ironic that the "O" network, home of countless bad made for TV movies, is also home to such a cool bunch of programmers.

Microsoft, OSS, and the Patterns and Practices Team

I've said it before, and I'll say it again now, the negative relationship between OSS efforts and Microsoft proper is an opportunity lost for the .Net community. Yes, I do think the lawyers are largely to blame for MS's consistent resistance to open source tooling. The Java world just uses JUnit & Ant inside of their IDE's (I'm thinking Eclipse &

IntelliJ). MS has to go and waste the time and energy to recreate NUnit and NAnt — among others. Those resources really could have been better spent on something else. But no, because of the constant threat of litigation, MS studiously avoids any relationship with OSS tools that originate from outside Redmond.

A side conversation that I had after the panel is that innovation in the .Net world is almost entirely limited to our one single vendor — Microsoft. OSS efforts, even when they're arguably more effective and higher quality, just can't make any inroads against MS's official position. In the OSS panel I brought up the Java community's far greater efforts in the OSS arena. Yes, the Java guys have more fragmentation, but they also have far many more choices and innovation comes from the community themselves. I mentioned to Ted Neward after the panel that I wanted very badly to jump into Ruby development last year when I was changing jobs because the Ruby community just has so much more energy and passion than .Net. Alas, being a .Net developer just flat out pays too much, and there's cool stuff coming down the turnpike for us without jumping ship (funny, I wrote this sentence before Hanselman's report from RailsConf).

Ayende took a shot yesterday at the Patterns and Practices team at Microsoft and the Composite Application Block that I thought was a little bit harsh. Chris Holmes fired back and called him out it. Ayende responded. Udi wrote a very reasoned opinion piece about the same subject called Why I don't worry about the Patterns and Practices Efforts. I'm going to come in somewhere between Ayende and Udi.

In our OSS in the Microsoft World panel the P&P team inevitable came up, since their work borders on open source and much of their work directly competes with OSS solutions. Both Ayende & I made some criticisms of the Patterns and Practices team and their work. I think the P&P team's framework products are hampered by the fact that these tools are written in a vacuum outside of any one project or group of projects. It's speculative design. I'm very strongly in the camp that says frameworks should be harvested and grown, not built. Another thing, it's hard to create innovative ideas on demand the way P&P probably has to work. In Mario Cardinal's session on Test Driven Architecture, he talked about "A ha!" moments where a sudden design realization leads to leaps in design quality. It's similar to Eric Evan's "breakthrough" idea from DDD. I guess all I'm saying is that the P&P has a very difficult job in some respects.

I kind of wish the P&P team wasn't necessary, but they are. Like it or not, they have a vastly more visible pulpit to teach design principles and push better tools than any one of us here in the blogosphere. I'm not wild about the tools that they produce and I've almost completely managed to avoid using them, but their work on explaining the underlying patterns of the tools is far more important than the tools themselves.

I do wish the P&P team would reach out much more to the OSS community. ObjectBuilder in particular probably could have been a better tool if they'd reached out and asked questions or at least researched existing tools instead of recreating the wheel. I'd also like to see the P&P team occasionally feature OSS tools. I.e., if you like Object-

Builder, try Castle Windsor, StructureMap, Spring.Net, etc.

I'll leave you with this thought (from me):

The Patterns and Practices team is a little beacon of Agile light in the Microsoft sea. Yes I don't think that many of the P&P solutions are really all that great, but I want the P&P team to succeed. A strong, visible P&P team benefits us greatly. The P&P team is making it easier for me to use the techniques like Dependency Injection and TDD on client engagements. That's a win.

You don't need to take the CAB

I might have helped spawn the Ayende/Chris Holmes thread on the CAB with this bullet point taken from my talk on WinForms design patterns – "Psst. You don't need the CAB to build large, maintainable WinForms applications." I meant that statement very literally, but even if you're using the CAB it's important to understand the underlying patterns and concepts first. I know there was a feeling among many of the Agile track speakers that the core design concepts are far too frequently skipped en route to memorizing the API's. You'll be much more effective with the CAB itself if you understand what the CAB designers were trying to accomplish and why the screen responsibilities are split the way they are.

At the end of this, I just don't think the CAB holds much value for me. Granted I haven't used it in anger, or really studied it all that much. Part of that is because every time I look at the CAB I see all kinds of unnecessary complexity that results from trying to be so generalized. Any framework that requires me to wrap my application around the framework has to cross a very high bar of value added to make me use that framework. I sincerely think that I can accomplish a solid separation of concerns and even pluggability with my own code and an API who's semantics reflect my application in specific, not a CAB abstraction.

Chris Holmes asked the rhetorical question "how long would it take to duplicate the CAB yourself?" The answer is that you don't. You grow something during the execution of your user interface features to support your functionality. If you have a strong understanding of the underlying patterns like Model View Presenter, synchronization options, Application Controller and Application Shell, and Event Aggregator's, it's pretty likely that you can build something that works better for your specific application than the one size fits all CAB. Wiring pluggable UI elements together is relatively simple with the usage of an IoC tool.

And yes Chris, I'd much rather use my own fully-featured StructureMap tool than deal with the very limited ObjectBuilder DI tool embedded inside the CAB. Guilty as charged.

I thought my WinForms patterns talk went well enough to get reused, but if I get, let's say 3+ comments asking for it, I'll convert that content to a set of blog posts along a "Build Your Own Customized CAB in 24 Days" theme to try to prove that building your own MVP infrastructure is a suitable task for mere mortals. The sample code is mostly written, so it's not that much work.

What's important?

About midway through DevTeach I realized that I hadn't managed to attend any of the sessions on newer technology. I wish I'd managed to get to the WF/WCF/WPF talks, but I'm not too concerned because the content that I saw was very good (and besides, I was heads down doing last minute code samples for my talks). The talks that I *did* attend were largely on subjects that I think are more core than API's. Configuration management, what to test and how, design patterns, design concepts, things that apply to every single project you'll undertake. Yes Linq and WCF are sexy, but I bet it won't improve ROI as much as improving your knowledge of software design in general.

C# 2.0 today is starting to feel like VB6 in 2001

In the spring and summer of 2001 I was working on a system that included a configurable workflow engine being coded in VB6. I pushed VB6 to its very limits trying to make it behave like a fully OOP language. VB6 felt limiting. C# came along with full OO and multi-threading. I felt empowered. Fast forward to today, and C# 2.0 is feeling limiting to me. I'm wanting to write more fluent interface code, and C# will do it, but it's bulky compared to other languages. I'm wanting more functional language support, but the 2.0 anonymous delegate syntax is far, far too verbose. I'm finding all kinds of uses for metaprogramming, which is far beyond C#'s abilities. Ayende mentioned wanting "missing method" type effects in .Net. It's time to move on. C# 3.0, or better yet, IronRuby, just cannot get here fast enough for my taste. Give me sharp tools thank you, and leave the safety scissors for someone else.

Challenging the Notion that Visual Programming is Good

I think the .Net community as a whole has a dangerous set of blinders on when it comes to visual programming. My nameless architect ended his talk with a quick demonstration of MS's new tools for defining a DSL modeling language for your tool. If you haven't seen it, it gives you a graphical design time tool that allows you to create UML like models that generate code. He made a whopping assumption that was taken in hook, line, and sinker by the audience — it's better because it's graphical. Is it really? And compared to what? For far too long, from Visual Basic, we in the Microsoft camp have worked on the assumption that visual tools and better design time support automatically equates to productivity. For our own good, we need to collectively challenge this assertion about the supposed superiority of visual programming and learn about code centric techniques that might lead to better productivity and far better maintainability.

ThoughtWorks is Busy

From a distance, I'm enthusiastic about some of the things coming out of ThoughtWorks right now as they seem to be reinventing themselves just a little bit (NOT excited enough to even consider going back and traveling every single week though). While I was at DevTeach I got to see a very brief demonstration of Mingle, the new Agile project management tool from ThoughtWorks Studio. Jeremy's in depth review from that 10 minutes — Mingle rocks! It mimics the Information Radiator motif you get from a physical representation in an electronic form. I'm using Jira & Confluence at my client, and I've used early versions of TargetProcess (for about a week before we blew it off), but Mingle already feels more "right" to me (but I am ex-TW and it's built for their style of working, so take that with a grain of salt).

Just as an aside, Mingle is written in Ruby on Rails, but running on JRuby. Do you remember when .Net was the multi language, single platform virtual machine, and Java was the single language, multi platform vm? It's looking more and more like each VM is going into more and more directions. That can't help but be better for us all in the long run.

JP gave ThoughtWorks studio a bit of link love too.

Entity Framework

I'll say this very briefly because I've already written negatively about the Entity Framework, and nothing substantive has changed since then. The Entity Framework is clearly a challenged project right now, and it's obvious that they're getting pulled in a lot of different directions, most of which I simply don't care about. When we went to the EF focus group it was immediately hijacked by the data centric guys as a forum to vent over EF's support for sprocs. Since I largely zoned out when they said Stored Procedure, you'll have to research what they were talking about on your own. The point being that they aren't going to be able to make everyone happy, and I think it's pretty likely that other voices will be much louder than mine when it comes to the feature prioritization of EF.

NHibernate just celebrated a new release. I'm thinking more and more about how to use iBatis for places where the database either stinks or you need to utilize the database capabilities. Because of the wonders of OSS, plus some solid commercial tools for persistence as well, I'm just not all that concerned about whether or not the Entity Framework ever ships right now. Unless a client forces me to use it just because it's from MS of course, then I'm going to care:(

Ok that's enough. I'm going in to work this morning for the first time in a week. I'm expecting to be just a little bit buried for awhile.

2 Preamble

Yesterday I made a somewhat unsubstantiated claim that you simply don't need the Composite Application Block (CAB) to build nontrivial WinForms user interfaces in a maintainable fashion. I claimed that mere mortal developers can master the underlying design patterns in the CAB and pick up a better Inversion of Control/Dependency Injection tool to organically grow an application structure that meets the specific needs of that application. I even think that mere mortals can do this as or more efficiently as they can by utilizing the very generalized CAB. To try to substantiate that claim, and because there seemed to be enough interest to warrant this series, I'm going to convert my WinForms patterns talk from DevTeach into a series of blog posts to demonstrate some design strategies for creating maintainable WinForms code. Even if you're going to stick to the CAB (and that is a valid choice), I can hopefully add to your ability to use the CAB by exploring the very same patterns you'll find lurking underneath the covers in the CAB. If you like the way the CAB does something better than the ways I present, please write in and say so.

One of my favorite authors is Alexandre Dumas, author of the Three Musketeers books and the Count of Monte Cristo. The Three Musketeers was the soap opera of its day, written in installments as a serial in whatever passed for magazines in those days. Likewise, I'm going to try to dribble these posts out in little installments. Basically the size of post I can write in a single sitting on the train ride from and to work.

Note on terminology — I'm a card carrying Fowlbot, so I'm using the terms set forth in Martin's forthcoming book on enterprise design patterns. **It's just the User Interface right?**

"Just" the user interface? I haven't seen this so much lately, but earlier in my career there was a definite attitude that User Interface (UI) programming was simple, grunt work — unworthy of the attention of the serious developer. In my mind I always think of the line "Just pick it up" from Caddyshack, or Kramer extolling Jerry to just "write it off." That's a dangerous attitude because user interface code can easily be much more complex than the server side code in many systems. It's worth spending some mental cycles on the design and structure of the UI code. Here's why:

- Creating user interface code is time intensive. Not just in terms of the initial coding, but in my experience UI code generates the most defects and hence takes the most time to fix bugs. We can cut down the defects by extending unit testing as far into the user interface code as possible. TDD opponents often use the UI as an example of code that just can't be tested. They're wrong, but UI testability doesn't come for free. It takes some care to craft a structure that makes UI code easier to test.

- User interface code interfaces with an extremely unpredictable external dependency — those wacky users! It takes a great deal of care to protect the system from garbage input and user error.

- The User interface changes frequently. Again, my experience is that it's much easier for a user and analysts to get the backend logic requirements upfront than it is to specify the user interface. Making the UI code easier to change just makes sense, but that again takes some design cycles.

- UI code is largely event driven, and event driven code can be nasty to debug. Again, testability and a cleaner separation of concerns makes that debugging either go away or at least become easier.

A Note about Design Patterns

I know many people blow off design patterns as ivory tower twaddle and silly jargon, but I think they're very important in regards to designing user interface code. Design patterns give us a common vocabulary that we can use in design discussions. A study of patterns opens up the accumulated wisdom of developers who have come before us. Finally, the design patterns we're going to examine in this series give us a framework of ideas on the best way to divide responsibilities in our user interface code. That's a good thing because... **User Interface code contains a lot of different responsibilities**

It was almost comical, but sitting in the Agile track at DevTeach almost every speaker, including me, had at least one slide dedicated to the Single Responsibility Principle (SRP). Just to recap, the SRP states that any one class "should have only one reason to change." Basically, you should strive for each class to contain a single coherent responsibility or concern. Like I said before, UI code can easily become complex. Just to prove that point, let's list some of the responsibilities that could easily be contained in a single screen:

- The actual presentation to the user
- Capturing to user input events
- Input validation and business rules
- Authorization rules
- Customization
- Screen flow and synchronization
- Interaction with the rest of the application

Lots of important things. Lots of things that would really be easier to code if I could just work on one thing at a time. Things I'd really like to be able to test in isolation from one another. Back to the Single Responsibility Principle, we can better enable change and improve maintainability in our UI code by assigning each of these responsibilities to distinct, loosely areas of the code. What we need is some sort of divide and conquer strategy for our WinForms code.

Of course, a large problem in WinForms development today is that the most common pattern of assigning responsibilities is... **The Autonomous View**

You might not be running around saying "I'm building an application on the Autonomous View pattern," but you'll recognize it in a heartbeat. The autonomous view is a single piece of code that encapsulates both presentation and screen behavior in one single lump of code. In WinForms the autonomous view is characterized by completely self-contained Form and UserControl's. While it's perfectly suitable for simpler screens, it can present some maintainability issues for screens of even moderate complexity because of the mixture of concerns. The biggest issue for me is that the code in an autonomous view is significantly harder to unit test than code in POCO classes. It's also potentially harder to understand because so many concerns are intermingled with the WinForms machinery. To some degree you can instantiate WinForms classes in memory and run unit tests against this code with or even without NUnitForms, but it's often much more trouble than it's worth.

To explain what I mean by the difficulty in testing, let's look at this example..,

To be continued in Part 2: Humble Views

3 The Humble Dialog Box

When last we left our hero, D'Artagnan was chasing the evil Lady de Winter across the breadth of France trying to intercept her on her dastardly mission when he found himself beset by disparate responsibilities within the tight confines of a single autonomous view with no room for sword play. D'Artagnan, being a perspicacious young man, quickly sees a way to separate the numerous concerns he's facing by opening his attack with...

The Humble Dialog Box

I'd say the very first concept to grasp in software design is separation of concerns. Divide and conquer. Eat the elephant one bite at a time. Learning how to decompose a bigger problem into a series of approachable goals. I'd rather work serially on a screen by completing one simple task before moving onto the next instead of working with all aspects of a screen in parallel. Division of responsibility for easier programming is a major consideration by itself, but there's another piece of motivation almost as important. Because user interface code can be very complex to debug and is prone to change based on user experience, I really, really want to extend granular unit testing with automated tests as far into the presentation layer as I possibly can.

Traditionally, user interface code has appeared to repel all but the most serious attempts at test automation. Automated testing against UI code was just flat out deemed too much work for the gain. That equation has changed over the last several years with the rise of architectures inspired by The Humble Dialog Box from Michael Feathers.

Here's the canonical example of the Humble Dialog Box I use to explain the concept. Say you have a user screen for some sort of data entry. You have a requirement that reads something like:

If the user attempts to close the XYZ screen without saving any outstanding changes, a message box is displayed to warn the user that they are discarding changes. If the user wishes to retain the outstanding work, do not close the screen. The message box should not be shown if the data has not been changed.

It's not that complex of a requirement really, but it's the kind of thing that makes a user interface client easy and convenient to use. We want this code to work. The code for it might look like this:

```
        private void ArrogantView_Closing(object sender, CancelEventArgs e)
        {
            // Let's not worry for now about how we figure out the screen has unsaved data
            if (isDirty())
            {
                bool canClose = MessageBox.Show"(Ok to discard changes or cancel to keep "working) ==
                    DialogResult.OK;
                e.Cancel = !canClose;
            }
        }
```

That code really isn't that complex, but let's think about how we could automate a test for this requirement to run within our regression test suite. That little bitty call to MessageBox.Show() and the modal dialog box that results is a serious pain to deal with in an automated test (it is possible, and I've done it before, but I'd strongly recommend you keep reading before you run off and try it). Observing the UI getting closed or not is also tricky, but I think the worst part is that to test this logic you have to fire up the UI, navigate to the screen, change some data on the screen, then trigger the close screen request. That's a lot of work just to get to the point at which you're exercising the code you care about.

Now, let's rewrite this feature as a "Humble" view, but before I show the new code, let's talk about the Humble view philosophy. The first thing to do is to put the view on a diet. Any code in a WinForms UserControl or Form is almost automatically harder to test than it would be in a POCO. A Humble view should be the smallest possible wrapper around the actual presentation code. Going farther, I don't want implementation details of the view mechanics to leak into other areas of the code, so I want to hide the View behind a POCO-ish interface. All that being said, the abstracted interface for our View could look like:

```
    public interface IHumbleView
    {
        bool IsDirty();
        bool AskUserToDiscardChanges();
        void Close();
    }
```

The view is also "passive," meaning that it doesn't really take any actions on its own without some sort of stimulus from outside the view. I'll discuss handling user events in depth in a later chapter, but for now let's just say that the view simply relays user input events to somewhere else with little or no interpretation.

One of the goals of a Humble view is to separate responsibilities. As in most designs, we want to assign different responsibilities to different areas of the code. In this case, we want to pull behavioral logic out of the view and into non-visual classes. If we put the view itself on a diet and pull out anything that isn't directly related to presentation, that extra code that implements things like behavior and authorization rules has to go somewhere. In this case we're going to move those responsibilities into a Presenter class:

```
public class OverseerPresenter
{
    private readonly IHumbleView _view;

    public OverseerPresenter(IHumbleView view)
    {
        _view = view;
    }

    public void Close()
    {
        bool canClose = true;
        if (_view.IsDirty())
        {
            canClose = _view.AskUserToDiscardChanges();
        }

        if (canClose)
        {
            _view.Close();
        }
    }
}
```

In particular, look at the Close() method. Some user event causes a call to the OverseerPresenter.Close() method. Inside this method we check the "dirty" state of the IHumbleView member and potentially ask the user to discard changes before proceeding to close the actual view. It's just about the exact same code, only now we can write an automated unit test to express this logic — with just a little help from our good friend RhinoMocks.

```
[TestFixture]
public class OverseerPresenterTester
{
    [Test]
    public void CloseTheScreenWhenTheScreenIsNotDirty()
    {
        MockRepository mocks = new MockRepository();
        IHumbleView view = mocks.CreateMock<IHumbleView>();
        Expect.Call(view.IsDirty()).Return(false);
        view.Close();
        mocks.ReplayAll();
        OverseerPresenter presenter = new OverseerPresenter(view);
        presenter.Close();
        mocks.VerifyAll();
    }

    [Test]
    public void CloseTheScreenWhenTheScreenIsDirtyAndTheUserDecidesToDiscardTheChanges()
    {
        MockRepository mocks = new MockRepository();
        IHumbleView view = mocks.CreateMock<IHumbleView>();
        Expect.Call(view.IsDirty()).Return(true);
        Expect.Call(view.AskUserToDiscardChanges()).Return(true);
        view.Close();
        mocks.ReplayAll();
        OverseerPresenter presenter = new OverseerPresenter(view);
        presenter.Close();
        mocks.VerifyAll();
    }

    [Test]
    public void CloseTheScreenWhenTheScreenIsDirtyAndTheUserDecidesNOTToDiscardTheChanges()
    {
        MockRepository mocks = new MockRepository();
        IHumbleView view = mocks.CreateMock<IHumbleView>();
        Expect.Call(view.IsDirty()).Return(true);
        Expect.Call(view.AskUserToDiscardChanges()).Return(false);
        // No call should be made to view.Close()
        // view.Close();
        mocks.ReplayAll();
        OverseerPresenter presenter = new OverseerPresenter(view);
        presenter.Close();
        mocks.VerifyAll();
    }
}
```

So I know what you might be thinking, what have I really gained here? Let me try to answer this:

- Orthogonality. We've moved behavioral logic out of the actual view. We can change the presentation or the behavior independently. That is a big deal.

- The screen behavior is easier to understand. I'm going to argue that this is a case of Reg Braithwaite's Signal to Noise Ratio in code (basically, expressing the intent of the code with little code that isn't directly related to the intent). When I want to understand the screen behavior, that behavior is the only signal I care about. Seeing (object sender, CancelEventArgs e) everywhere in the middle of the behavioral code is noise. The converse is true as well when I'm working on the presentation itself.

- The screen behavior is easier to test and modify. That's enough by itself to justify the Humble View style. What if this closing behavior changes tomorrow with a requirement to set a user preference to never ask users to discard changes? If I'm working in a Humble View style, I can probably make that screen behavior change completely, including unit tests, in the Presenter class by itself without ever having to fire up the user interface until the very last sanity check. Verifying little behavior changes with NUnit is a far, far tighter feedback cycle than doing the save verification by firing up the user interface and doing the manual input steps necessary to exercise the functionality. Tight feedback cycles == productivity.

- By extending the reach of granular and automated unit tests farther into the potentially complex user interface code, we can drastically slow down the rate of screen defects getting through to the testers. If nothing else, we can knock down all the common uses of the user interface quickly through tests to give the testers more time to break the application with edge cases and exploratory testing.

An astute reader will note that we didn't write any unit tests for the View. I'll show an example in later chapters of testing the View itself, but the philosophy in general is to make the hard to test view code so simple as to be reliably verified by inspection alone. I.e., you should make the View code so simple that it's almost hard to screw it up.

A Taxonomy of Humble Views

Arguably the first Humble View is the original Model View Controller architecture handed down, as basically everything good in software developer seems to be, from the Smalltalk community. As I've mentioned before, you can read about the evolution of the Model View Controller (MVC) pattern and the formulation of the Model View Presenter (MVP) patterns that we're mostly talking about in this series as user interface toolkits changed.

D'Artagnan's masterful implementation of the Humble Dialog Box vanquishes his foes, but he knows he'll need trusted companions now that the evil Lady de Winter surely knows he is in pursuit. Fortunately, D'Artagnan's trusty three companions are riding hard to join him. D'Artagnan smiles to himself and imagines his friends coming around the bend in the road:

- Supervising Controller — I'm here to help the View with the harder cases!

- Passive View — I only do what I'm told

- Presentation Model — Just do what I do

D'Artagnan sees dust rising in the air, a traveler is coming...

To be continued in Part 3: Supervising Controller

QUICK NOTE: I showed the MessageBox stuff happening as a consequence of calling a method on the IView interface. In the past I've also used some sort of IMessageBox-Creator interface to create message boxes directly. There are advantages either way.

4 Supervising Controller

When last we left D'Artagnan he had just concluded a successful duel with a number of screen concerns by dividing them with a masterful usage of the Humble Dialog Box. As D'Artagnan strives to regain his breathe, he's heartened by the appearance of his three doughty companions Athos, Aramis, and Porthos. As the four friends sit down in a shady spot besides the road for a fine meal of delicacies (unknowingly donated by a noblewoman of Athos' acquaintance), D'Artagnan describes their predicament when they manage to corner the Lady de Winter and her host of minions. **A Shipping Screen**

Let's imagine that you need to build a little screen to allow a user to configure the shipping options for some sort of electronic order that looks something like this screenshot below:

The user first needs to select the state or province (since this was for a talk in Canada, I thought it would be nice to let our Canuck neighbors to the north use the app too) that is the destination of the shipment. Not every shipping vendor can ship to every destination, so we should modify the contents of the shipping vendor dropdown to reflect the destination. Likewise, a change in the selected shipping vendor will also cascade down to the list of shipping options (think Overnight, 2 business day, parcel, etc.). Some, but not all, shipping options allow the customer to purchase insurance in case of loss or require a signature at the destination for guaranteed delivery. Finally, the cost of the shipment should be recalculated and displayed on the screen anytime the shipping selections are changed.

I can spot a couple different responsibilities in just that paragraph, plus a couple more down the line when it's time to actually submit the shipment. Even this little screen has enough complexity in it that I wouldn't want to bundle all of the responsibilities into a single autonomous view. So let's enumerate just the responsibilities from that paragraph and start thinking about how to assign these responsibilities to different classes.

- Display and capture the currently selected shipment options
- Respond to user events like selecting a value in the ComboBox's
- Fetching the list of states
- Fetching the list of shipping vendors for a given state or province

- Fetching the list of options for a given state/province and vendor
- Changing the dropdown lists
- Enabling and disabling the checkbox's for purchasing insurance and requiring a signature
- Calculate the shipping cost for the selected options
- Update the cost textbox whenever the shipment options change

That's a fair amount of behavior and business logic for one little screen. We know that we probably want to employ some sort of Humble View approach to split out some of the responsibilities from the Form class and into POCO classes that are easier to test. I'm going to show three sample solutions of the shipping screen, each using a bit different approach to assigning responsibilities.

Interlude

The four friends mull over the looming fight with the forces of Lady de Winter as they finish off a fine cheese produced from the endless saddlebags of Aramis's manservant. D'Artagnan poses this question to the older musketeers: "When I'm beset by a multitude of concerns in a single screen, what technique should I use to best each concern in turn?" The three older musketeers ponder the question raised by their younger companion. Finally, Athos speaks up. "I would not concern myself with the simpler responsibilities of a screen. I would first seek to eliminate the more complicated screen responsibilities by employing…"

The Supervising Controller

The goal of the Supervising Controller variant of Model View Presenter is to remove the more complicated screen scenarios that deserve more unit testing attention out of the view and into a separate Presenter class. The Supervising Controller strategy takes advantage of the data binding support in WinForms for simple screen synchronization. To that end, we'll create a simple class called Shipment that will be our Model in the MVP triad.

```
    public class Shipment : INotifyPropertyChanged
    {
        private string _stateOrProvince;
        private string _vendor;
        private string _shippingOption;
        private bool _purchaseInsurance;
        private bool _requireSignature;
        private double _cost;

        public string StateOrProvince
        {
            get { return _stateOrProvince; }
            set
            {
                _stateOrProvince = value;
                fireChanged"("StateOrProvince);
            }
        }

        public string Vendor
        {
            get { return _vendor; }
            set
            {
                _vendor = value;
                fireChanged"("Vendor);
            }
        }
        // And the rest of the …properties
  }
```

The View itself simply has a setter that takes in a Shipment object and starts up the data binding.

```
        public Shipment Shipment
        {
            set
            {
                // start up the data binding stuff
            }
        }
```

I'll talk about options for the Model in depth in a later post, but for now, let's say that Shipment is just a dumb batch of getters and setters. Since I can't stand writing INotifyPropertyChanged interface implementations by hand, you'll probably want to codegen these Model classes — again giving me even more reasons to keep the Shipment class dumb.

We've taken care of the actual presentation of the shipment data, so let's move on to more responsibilities. There's no possible way that a screen should know how to calculate the shipping costs, and probably shouldn't know how to fetch the data for the dropdown selections. Deciding whether or not a shipper and shipping option allows a user to purchase insurance or require a signature on receipt is business logic for the real domain logic classes, not screen logic that belongs in the screen. Let's not particularly worry about how this stuff is implemented right now. Let's just define an interface for all of this functionality (and a Data Transfer Object as well).

```
public class DeliveryOptions
{
    private bool _purchaseInsuranceEnabled;
    private bool _requireSignatureEnabled;

    public bool PurchaseInsuranceEnabled
    {
        get { return _purchaseInsuranceEnabled; }
        set { _purchaseInsuranceEnabled = value; }
    }

    public bool RequireSignatureEnabled
    {
        get { return _requireSignatureEnabled; }
        set { _requireSignatureEnabled = value; }
    }
}

public interface IShippingService
{
    string[] GetLocations();
    string[] GetShippingVendorsForLocation(string location);
    string[] GetShippingOptions(Shipment shipment);
    void CalculateCost(Shipment shipment);
    DeliveryOptions GetDeliveryOptions(Shipment shipment);
}
```

The IShippingService interface works on our Shipment class that we're binding to in the actual screen, as opposed to exposing primitive arguments. When I was writing the sample code it seemed to me to be a simple way of interacting with the service because it cuts down on any data transformations between screen and service. The CalculateCost(Shipment) method would also write the cost back to the Shipment object, cascading a corresponding change to the screen as the data binding updates the screen based on changes to Shipment. It's probably worth noting that this IShippingService could just be a Facade class over the business logic specifically created for easier consumption by the user interface. In the next chapter I'll take a different approach that exposes the underlying Domain Model classes for the shipping system.

At this point we're largely left with just responsibilities for changing the dropdown options and mediating between the View and the IShippingService. That's where the ShippingScreenPresenter finally comes in to provide the missing functionality that goes beyond simple data binding, as well as coordinating user actions with the IShippingService.

```
public class ShippingScreenPresenter
{
    private readonly IShippingScreen _view;
    private readonly IShippingService _service;
    private Shipment _shipment;

    public ShippingScreenPresenter(IShippingScreen view, IShippingService service)
    {
        _view = view;
        _service = service;
        _shipment = new Shipment();

        // Since we're got the INotifyPropertyChanged interface on Shipment,
        // we might as well use it to trigger updates to the Cost
        _shipment.PropertyChanged += new PropertyChangedEventHandler(_shipment_PropertyChanged);
    }

    private void _shipment_PropertyChanged(object sender, PropertyChangedEventArgs e)
    {
        _service.CalculateCost(_shipment);
    }

    public Shipment Shipment
    {
        get { return _shipment; }
    }

    public void Start()
    {
        _view.Shipment = _shipment;
    }

    // React to the user selecting a new destination
    public void LocationChanged()
    {
        _view.Vendors = _service.GetShippingVendorsForLocation(_shipment.StateOrProvince);
    }

    // React to the user changing the Vendor
    public void VendorChanged()
    {
        _view.ShippingOptions = _service.GetShippingOptions(_shipment);
    }

    // React to the user changing the Shipping Option
    public void ShippingOptionChanged()
    {
        DeliveryOptions options = _service.GetDeliveryOptions(_shipment);
        _view.InsuranceEnabled = options.PurchaseInsuranceEnabled;
        _view.SignatureEnabled = options.RequireSignatureEnabled;
    }
}
```

Since I've been claiming that this style of user interface structure improves the testability of the screen as a whole, let's take a look at what the unit tests might look like. Here's the unit test for correctly enabling or disabling the insurance and signature checkbox's on the shipping screen after the shipping option changes:

```
        [Test]
        public void EnableOrDisableTheInsuranceAndSignatureCheckboxesWhenShippingOptionChanges()
        {
            MockRepository mocks = new MockRepository();
            IShippingService service = mocks.CreateMock<IShippingService>();
            IShippingScreen screen = mocks.CreateMock<IShippingScreen>();
            ShippingScreenPresenter presenter = new ShippingScreenPresenter(screen, service);

            // Setting up the expected set of Delivery options
            DeliveryOptions deliveryOptions = new DeliveryOptions();
            deliveryOptions.PurchaseInsuranceEnabled = true;
            deliveryOptions.RequireSignatureEnabled = false;

            // Set up the expectations for coordinating both
            // the service and the view

            Expect.Call(service.GetDeliveryOptions(presenter.Shipment))
                .Return(deliveryOptions);
            screen.InsuranceEnabled = deliveryOptions.PurchaseInsuranceEnabled;
            screen.SignatureEnabled = deliveryOptions.RequireSignatureEnabled;

            // Perform the work and check the expectations
            mocks.ReplayAll();
            presenter.ShippingOptionChanged();
            mocks.VerifyAll();
        }
```

The first thing you might notice is that this is definitely an interaction based unit test. That's not surprising since one of the primary duties of a Presenter is to mediate between the services and view. In development with Test Driven Development / Behavior Driven Development, it's often advantageous to separate the responsibility for performing an action away from the decision to perform that action. In this case, the real View enables the insurance and signature checkboxes when the Supervising Presenter tells it to enable or disable the checkboxes. In other words, the Supervising Presenter is the mostly immobile queen bee, and the View is the mobile, but relatively brainless, worker bee. In unit tests like the one above, we're largely testing that the Presenter is sending the correct signals to the View and service interfaces.

For another example, here's a unit test for recalculating the shipment cost as the selected shipping options change:

```
        [Test]
        public void UpdateTheCostWheneverTheShipmentChanges()
        {
            MockRepository mocks = new MockRepository();
            IShippingService service = mocks.CreateMock<IShippingService>();
            IShippingScreen screen = mocks.CreateMock<IShippingScreen>();
            ShippingScreenPresenter presenter = new ShippingScreenPresenter(screen, service);
            service.CalculateCost(presenter.Shipment);
            mocks.ReplayAll();
            presenter.Shipment.Vendor = "a different "vendor;
            mocks.VerifyAll();
    }
```

That wasn't that bad, now was it? But wait, you ask. Where's the actual logic for calculating the shipment cost? For right now I'm just worried about the wiring of the screen itself. Yes, this unit test covers very little ground, and it's not a "real" test, but I've created some level of trust that this particular linkage in the code does work. I can now turn my back on the screen itself and test the actual ShippingService by simply pushing in Shipment objects and checking that the Cost field is correctly updated. Not one single line of the shipping screen code needs to be present for any of that logic to be tested. I hope it's needless to say that there really shouldn't be any leakage of domain logic into the screen code. Real domain logic in views is just about the fastest way possible to create an utterly unmaintainable system.

Interlude

The four friends pondered Athos's solution in quiet contemplation in the state of contentment that only follows a fine meal. “Wait,” exclaims D'Artagnan, “You haven't told us how the View talks to the Presenter, and who creates who! How should this work?” Athos simply shakes his head and says “it's a long ride to Dunkirk, we'll talk more of this on the road.” (as in, I'll get there, just give me a couple more chapters – Jeremy).

How I Prefer to Work

Personally, I wouldn't order the work quite the way I showed above. I like to start with either a screen mockup or maybe even the actual view. If you start with the actual concrete view class, don't wire it up yet, and whatever you do, don't automatically implement the intended “IView” interface until you're done with the Presenter. Sometimes I'll start by taking some notes or sketching out some UML or CRC models about the screen scenarios. My next step is to encode the behavioral aspect of the screen in the Presenter. I use RhinoMocks in place of both the View and whatever service interfaces the Presenter needs. As much as possible, I like to define the methods and interactions of the Presenter with the services and view in a unit test body and just let ReSharper generate the methods on the dependencies. Since we're strictly dealing with interfaces here, we don't have to break our flow with the Presenter to implement the services and view just yet. As soon as the Presenter itself is fully fleshed out, I implement the IView interface on the real View and create an implementation for any of the service dependencies.

This probably isn't realistic for complex screens, but at least on simple screens it should become quite ordinary for a screen to just work the first time it's run in the UI — assuming you unit tested all of the pieces individually.

Summary

I'd bet that Supervising Controller is probably the most commonly used Humble View architecture, but I don't have any figures at hand on that. It still allows you to use the design time support to layout controls and even configure the data binding with the visual tools (but we're going to talk about options to data binding later). My current project is using Supervising Controller (sans data binding), and I think it's largely where the sweet spot is. I will use the other options at times though, and there are a lot of different variations on each of the other Humble Views, so we're not done yet.

Is this really worth doing? I say yes, otherwise I wouldn't be doing it. Let me put it to you this way, on the first project I used the Humble View (Passive View actually) approach, we saw a strong correlation between the bug count per screen and the unit test coverage for the code on that screen. Screens that were driven with unit tests on the Presenter had many fewer behavioral bugs. I'm talking it even farther on my current project by adding NUnitForms tests on the View to Presenter interaction, plus Fit tests on the actual screen behavior.

If you're not a fan of interaction based testing, and using mock objects gives you an allergic reaction, don't worry. I've got stuff for you too coming up.

Conclusion

The four friends pondered the wisdom of Athos' approach while drinking their way through the rest of the wine from D'Artagnan's packs. Suddenly, mighty Porthos clears his throat and…

To be continued in Part #3 – The Passive View Pattern

5 Passive View

When last we left our brave companions, between courses of cheese and fine wine, Athos was sharing his strategy for dividing screen responsibilities by employing the Supervising Controller pattern. Mighty Porthos suddenly cleared his throat and exclaimed "since I am the strongest man in all of France, I would face my opponents a different way. Because the View itself is the trickiest piece of code to test and maintain, I would squeeze the View with great force until the only thing left inside the View is a mere skeleton of presentation logic. I will render the Lady de Winter's greatest warrior a mere… **Passive View**

Last time we looked at a small screen that allows a user to select options for configuring the shipment options for some sort of online order. We examined a sample approach utilizing the Supervising Controller variant of Model View Presenter that left the View in charge of simple screen synchronization while utilizing an external Presenter class to handle more complex behavior and all communication with the rest of the system. This time around we're going to build the exact same system, but use the Passive View variant of Model View Presenter.

It's probably easiest to explain Passive View by first explaining how it's different than Supervising Controller. The goal of the Passive View is to maximize the ability to automate testing of the screen, and that means taking as much as possible out of the hard to test View code and moving it to the Presenter. For that reason, the biggest difference is the reduced role of the View — even from the already slimmed down View attached to a Supervising Controller. The Presenter/Controller is responsible for all screen synchronization. The View in Passive View is an extremely thin wrapper around the presentation details with next to no behavior of its own. The view probably doesn't even know about the Model classes. To start the Passive View solution, let's look first at the interface for IShippingScreen:

```
public interface IShippingScreen
{
    string[] ShippingOptions { set; }
    string[] Vendors { set; }
    bool InsuranceEnabled { set; }
    bool SignatureEnabled { set; }
    string StateOrProvince { get; set;}
    string Vendor { get; set;}
    string ShippingOption { get; set;}
    bool PurchaseInsurance { get; set;}
    bool RequireSignature { get; set;}
    double Cost { get; set;}
}
```

As you can probably guess from this interface alone, the View becomes simplistic. The Presenter is now responsible for telling the view what piece of information to put into each UI widget. The concrete View is going to end up looking something like this:

```
public partial class ShippingScreen : Form, IShippingScreen
{
    public ShippingScreen()
    {
        InitializeComponent();
    }

    public string ShippingOption
    {
        get { return shippingOptionField.SelectedText; }
        set { shippingOptionField.SelectedText = value; }
    }

    public bool PurchaseInsurance
    {
        get { return purchaseField.Checked; }
        set { purchaseField.Checked = true; }
    }

    // Who sees a problem here?
    public double Cost
    {
        get { return double.Parse(costField.Text); }
        set { costField.Text = value.ToString(); }
    }
    // Stuff that we don't care about
    #region Stuff that we don't care about
    #endregion
}
```

At this point, the view is as dumb as we can possibly make it. It should be nearly trivial to verify the functioning of the View code by simple inspection — for the most part anyway. We could easily decide to forgo testing the actual View itself at this point and judge that to be a perfectly acceptable compromise.

The Presenter is now more complicated because it's taking on the additional responsibility for synchronizing data between the screen and the Domain Model. In this case it's making a one to one transference of properties from the screen elements to the properties of the Shipment class.

```
    private Shipment createShipmentFromScreen()
    {
        Shipment shipment = new Shipment();
        shipment.PurchaseInsurance = _screen.PurchaseInsurance;
        shipment.StateOrProvince = _screen.StateOrProvince;
        // so on, and so forth
        return shipment;
    }
```

For a small screen, that's not that bad. In reality, I generally use Passive View for small screens like login screens. I do find the screen synchronization to be a chore. Then again, think about the case of a screen that serves to display and edit an aggregate object structure with multiple levels of hierarchical data. Data binding works best with "flat" objects, so you've got to do something. In my mind, sacrificing the structure of the business domain to fit the user interface tooling is mostly a poor compromise. To have the best of both worlds you can either create a wrapping object to provide a "flattened" view of the object hierarchy to allow data binding to work, or skip that and use Passive View to have better control over the screen synchronization.

What is cool about Passive View, besides the extra testability, is a further set of insulation between the presentation technology and the business and service layers of the application. When I built the Supervising Controller approach, I deliberately used an object specifically built for the data binding and hid the "real" domain behind IShippingService. This time around, let's let the Presenter work with the "real" domain classes somewhat. That being said, ShippingScreenPresenter will now interact with an IShipper class to get at the business rules for a particular shipping option.

```
public interface IShipper
{
    bool AcceptsInsurance { get;}
    bool CanRequireInsurance { get;}
    string[] Options { get;}
    string Description { get;}
    bool CanCaptureSignature { get; }
    double CalculateCost(Shipment shipment);
}
```

Of course, we've got to find the correct IShipper in the first place. For that, we'll use a Repository:

```
public interface IShipperRepository
{
    IShipper[] GetShippersForLocation(string location);
    IShipper FindShipper(string shipperName);
}
```

Even more so this time, the ShippingScreenPresenter is largely a Mediator between the interfaces exposed by IShippingScreen, IShipper, and IShippingRepository. The screen synchronization can be more work inside the presenter, but I think you can get by with less abstraction of the rest of the application. The ShippingScreenPresenter might look like this:

```
public class ShippingScreenPresenter
{
    private readonly IShippingScreen _screen;
    private readonly IShipperRepository _repository;
    private IShipper _shipper;

    public ShippingScreenPresenter(IShippingScreen screen, IShipperRepository repository)
    {
        _screen = screen;
        _repository = repository;
    }

    public void ShipperSelected()
    {
        _shipper = _repository.FindShipper(_screen.Vendor);
        _screen.ShippingOptions = _shipper.Options;
        _screen.InsuranceEnabled = _shipper.CanRequireInsurance;
        _screen.SignatureEnabled = _shipper.CanCaptureSignature;
    }

    private Shipment createShipmentFromScreen()
    {
        Shipment shipment = new Shipment();
        shipment.PurchaseInsurance = _screen.PurchaseInsurance;
        shipment.StateOrProvince = _screen.StateOrProvince;
        // so on, and so forth
        return shipment;
    }

    public void OptionsChanged()
    {
        Shipment shipment = createShipmentFromScreen();
        _screen.Cost = _shipper.CalculateCost(shipment);
    }
}
```

Needless to say, using the Passive View pretty well demands an Interaction Testing style of unit tests with lots of mock objects. If you don't grok RhinoMocks, Passive View might not be for you. Supervising Controller is a definite alternative, but in a later section I'll take a look at the Presentation Model pattern for a state-based style of unit testing. **Interlude**

Young D'Artagnan looks puzzled. He finally asks his older friends "Wouldn't that make the communication between the Presenter and View very chatty? And what's to stop the Presenter from becoming just as convoluted as an Autonomous View?" Porthos snorts and exclaims "You are a perspicacious lad! I would not stop with crushing the View into submission. I will use my superior strength to crush the Presenter until it only contains a single, cohesive responsibility!" **Summary**

I used Passive View quite extensively on my first WinForms project in 2004. Overall, the experience hooked me for life on using Humble Dialog Box design philosophies for building fat clients. It became routine for screens to work on the very first attempt to run new screen features — assuming that you really did unit test the individual pieces first. We also so another noticeable trend, screens that were fatter with more logic and less unit test coverage generated alarmingly more bugs and took much more time to debug.

I mentioned that the screen synchronization can become a chore. The presenter potentially picks up more responsibilities for screen synchronization and state like "IsDirty" checks. The downside of Passive View is a chatty interface between View and Presenter. I wrote an article early last year detailing my Best and Worst Practices for Mock Objects.

Most of the advice for what not to do with Mock objects came from that same project that we used Passive View. My advice is to watch the size of the Presenter. If it gets too big, split up responsibilities. Maybe you take screen synchronization, IsDirty logic, and maybe validation and put it in some kind of “inner” presenter. The “Inner” presenter talks to the View itself. The “Outer” presenter talks to the “Inner” presenter and the outside world.

One way to detect a need for this Inner/Outer presenter case is to watch your unit tests. If you ever find yourself writing an uncomfortable number of mock object expectations in any one test, you’re probably violating the Single Responsibility Principle (SRP). If you find yourself setting up mock object expectations for something that’s barely relevant to the subject of the unit test, you almost automatically split the class under test into multiple classes. One of the best design tricks you can apply is to continuously move your design closer and closer to SRP. Conclusion

The four friends finished their repast and sat around the fire, sated from the provisions generously supplied by a minor noblewoman of Atho’s acquaintance. Aramis, who the companions know is the craftiest of the four friends, speaks up: “slaying a View of many responsibilities with the sharp edge of a mock object is a fine thing, but I think that we can use our wits to fool the screen into subservience to state based testing by employing the Presentation Model to…”

6 Presentation Model

Our four friends are crafting a strategy for the inevitable and highly anticipated clash with the minions of the Lady de Winter. Mighty Porthos has just finished a long oration about his Passive View approach to creating maintainable WinForms screens. The crafty Aramis has started his own oratory on his preferred approach to avoid so much Interaction Based Testing by utilizing…

The Presentation Model

The Presentation Model differs from the two Model View Presenter approaches (Supervising Controller and Passive View) by combining the “M” and the “P” into a single class. This single Presentation Model class both contains the state of the screen and implements the behavior of the screen. Compared to the Supervising Controller, Presentation Model is more complex in that it also implements the state of the screen, but also less complex because the state synchronization is almost entirely the responsibility of the View itself. Instead of a Presenter directing the View to change the state of the screen, the Presentation Model simply changes its own state and depends on Data Binding (or an equivalent) to update the screen accordingly.

Let’s jump right into our third and final implementation of the Shipping Screen. As I stated before, the View will use data binding to bind its screen elements to the public properties on the new ScreenPresentationModel class.

```
    public partial class ShippingScreen : Form
    {
        public ShippingScreen()
        {
            InitializeComponent();
        }

        public void Bind(ShippingPresentationModel model)
        {
            shipmentBindingSource.DataSource = model;
        }
    }
```

I'll spare you the rest of the data binding setup code, and besides, that's covered in other literature to a vastly greater degree than Presentation Model. Besides which, I've barely worked with data binding and you've probably guessed correctly that I'm more than a little biased against it.

So far we haven't seen anything that different from Supervising Controller, but when you use the Presentation Model approach you're probably exploiting the fact that data binding in WinForms can also bind to the "Visible" and "Enabled" properties of controls and not just the value. In the case of the Shipping screen, we'll bind the "Enabled" properties of the checkbox's for selecting insurance and requiring a signature to properties on the ShippingPresentationModel shown below.

```
        public bool InsuranceEnabled
        {
            get { return _insuranceEnabled; }
            set { _insuranceEnabled = value; }
        }

        public bool SignatureEnabled
        {
            get { return _signatureEnabled; }
            set { _signatureEnabled = value; }
        }

        public string Vendor
        {
            get { return _vendor; }
            set
            {
                _vendor = value;
                fireChanged"("Vendor);
                DeliveryOptions options = _service.GetDeliveryOptions(this);
                InsuranceEnabled = options.PurchaseInsuranceEnabled;
                SignatureEnabled = options.RequireSignatureEnabled;
            }
        }
```

In the above code, anytime a user selects a different shipping vendor the data binding will call the setter for Vendor on ShippingPresentationModel, causing a recalculation of the InsuranceEnabled and SignatureEnabled properties, which finally causes the two checkbox's to be either enabled or disabled depending upon the shipping vendor selected. All because a little bug went kachooo!

The communication between View and the Presentation Model is relatively simple, the View simply sets properties on the PresentationModel class and cascading actions are triggered in the setters. I've purposely put off talking about View to Presenter communication, but let's just say that this aspect of the Presentation Model is simpler than either Supervising Controller or Passive View.

One last example of the ShippingPresentationModel. There are three or four factors that influence the cost of the shipment. If any of these screen elements change, the cost needs to reevaluated. With Presentation Model, I just capture all change events inside the setters, so the trigger to reevaluate the shipping cost is something like this:

```
        public string ShippingOption
        {
            get { return _shippingOption; }
            set
            {
                _shippingOption = value;
                fireChanged"("ShippingOption);
                _service.CalculateCost(this);
            }
        }

        public bool PurchaseInsurance
        {
            get { return _purchaseInsurance; }
            set
            {
                _purchaseInsurance = value;
                fireChanged"("PurchaseInsurance);
                _service.CalculateCost(this);
            }
        }

        public bool RequireSignature
        {
            get { return _requireSignature; }
            set
            {
                _requireSignature = value;
                fireChanged"("RequireSignature);
                _service.CalculateCost(this);
            }
        }
```

I simply make a call to IShippingService.CalculateCost(IShipment) anytime a property changes that impacts the shipping calculation. For convenience sake, I made ShippingPresentationModel implement a common IShipment interface that is consumed by IShippingService, if you're wondering where in the world the "this" parameter was coming from. I'm assuming that the IShippingService will itself set the IShipment.Cost property. The signatures for IShippingService still looks like this:

```
    public interface IShippingService
    {
        string[] GetLocations();
        string[] GetShippingVendorsForLocation(string location);
        string[] GetShippingOptions(IShipment shipment);
        void CalculateCost(IShipment shipment);
        DeliveryOptions GetDeliveryOptions(IShipment shipment);
    }
```

State Based Unit Testing

The biggest difference to me in using Presentation Model versus one of the MVP patterns is the shift to state based testing inside of our xUnit tests. I'm more or less a "mockist" I guess, but I've worked with more than a few people who've had almost allergic reactions to using mock objects. If you're one of those people, don't worry, you're not out of luck because you can do more or less state based testing with Presentation

Model. Here's an example of what I mean (even though out of pure contrariness I'm using RhinoMocks to create my stub):

```
        [Test]
        public void ResetTheShippingOptionsWhenTheStateOrProvinceIsChanged()
        {
            // We're going to test ShippingPresentationModel in a state-based manner
            // I'm using RhinoMocks to create the stub just because
            //   a.)  It's easy
            //   b.)  I hate cluttering up the code with static mocks and stubs if
            //        I don't have to.
            // You might note that I'm not even bothering to call mocks.VerifyAll()

            MockRepository mocks = new MockRepository();
            IShippingService service = mocks.CreateMock<IShippingService>();
            ShippingPresentationModel model = new ShippingPresentationModel(service);
            string[] theOptions = new string"[]{Option  1, "Option  2, "Option  3};
            Expect.Call(service.GetDeliveryOptions(model)).Return(theOptions).Repeat.Any();
            mocks.ReplayAll();

            // We need to verify that the model starts with a zero array string
            Assert.AreEqual(0, model.ShippingOptions.Length);
            // I'm not sure I'd bother testing the raising of the PropertyChanged event,
            // or at least do it in another test.
            bool propertyWasCalled = false;
            model.PropertyChanged += delegate (object sender, System.ComponentModel.PropertyChangedEventArgs
                e)
      {
        propertyWasCalled = e.PropertyName == ""ShippingOptions;
      };

            model.StateOrProvince = ""TX;
            // Check that the state of the model changed
            Assert.AreEqual(theOptions, model.ShippingOptions);
            // And while we're at it, let's check that the PropertyNotified event was called
            Assert.IsTrue(propertyWasCalled);
        }
```

All I'm testing here is that the ShippingPresentationModel gets a list of Shipping Options whenever the StateOrProvince property is changed, then resets its ShippingOptions property. I'm not real wild about it, but I also showed using an anonymous delegate to check that the PropertyChanged event was fired for "ShippingOptions." To recap, the expected sequence of events is:

The user selects a value in the State or Province select box. Data binding in the view sets the StateOrProvince property on ShippingPresentationModel. Since the View is just talking directly to getters and setters, we really don't need the View involved in this unit test at all. In the setter for StateOrProvince, the ShippingPresentationModel should find the ShippingOptions for the selected state or province and set it's internal value for ShippingOptions which... Fires the PropertyChanged event for "ShippingOptions" which directs the magical data binding support to fill the dropdown list for Shipping Options with new values (which I didn't show because it's documented very well on MSDN).

Whew. The code that implements this test above is much simpler:

```
        public string StateOrProvince
        {
            get { return _stateOrProvince; }
            set
            {
                _stateOrProvince = value;
                // Whenever this property changes, we need to reset the
                // ShippingOptions to match the StateOrProvince
                ShippingOptions = _service.GetShippingOptions(this);
                fireChanged"("StateOrProvince);
            }
        }

        public string[] ShippingOptions
        {
            get { return _shippingOptions; }
            set
            {
                _shippingOptions = value;
                fireChanged"("ShippingOptions);
            }
        }
```

Summary

The Presentation Model is another example of a Humble View. It largely differs from the Model View Presenter patterns by combining the Model and Presenter into a single class. It's important to note that the Presentation Model most likely encapsulates the actual application model, and it's definitely part of the user interface rather than a domain model class implementing pure business logic. While it does a great job isolating behavior from the View and exposing the behavior in a way that allows for state based testing, you might find yourself getting annoyed at all the delegation that has to take place between the Presentation Model class and the inner application model. Then again, a buffer between the user interface and the rest of the application might just be a good thing.

Honestly, I haven't used this pattern but a time or two. The largest implementation I've seen was actually a Java Swing client where it was used quite effectively.

I do have an example from StoryTeller that I will probably use in the posts on creating an Application Shell where I use Presentation Model as a kind of state machine to synchronize menu state as the screen mode changes. I'm leaving it out now for the sake of brevity (and my impending bedtime). Other Resources

I think Presentation Model is another name for the Model/View/ViewModel pattern being promoted by some of the WPF team at Microsoft. I still think I'd rather stick with Supervising Controller in most cases, but I'm thrilled that people in MS itself are talking about this at all. It's an old post, but I'd read Michael Swanson's thoughts on Presentation Model before you run off and use the pattern.

Three Musketeers Retirement Notice

The silly Three Musketeers thematic interludes just require more creativity than I can summon on a regular basis. Let's just say they stopped the evil Lady de Winter in her diabolical mission (even though she's usually the most interesting character in the movie adaptations. Seriously, who are you going to root for, Chris O'Donnell or

Rebecca De Mornay/Faye Dunnaway? That's what I thought;) and delivered their very complex WinForms application on time with minimal fuss with a healthy infrastructure of automated testing. And for the hard core Dumas fans, let's just forget about how badly things end in the Man in the Iron Mask because it still depresses me in a way that has only been topped by "they killed Wash!."

7 View to Presenter Communication

Alright, I've purposely hid the View to Presenter communication in my previous posts on Supervising Controller and Passive View because I thought that subject was worthy of its own post. As I see it, there are 2 1/2 basic ways to communicate screen events back to the Presenter.

Expose events off of the IView interface. Jeffrey Palermo has told me before that he will do something very similar, but attaches delegates through setter properties instead. I'm calling that technique the "1/2 because it feels very similar to me to using events. Let the View have a reference to the actual Presenter and just call methods on the Presenter inside of screen events.

Just taking a straw poll of the blog posts out there about this topic, the majority of the world seems to favor events. Just to be contrary, I very strongly prefer direct communication and it's time for that approach to get some love .

Doing it with Events

Making the View to Presenter communication work through events has the indisputable advantage of maximizing loose coupling. The View doesn't even need to know that there's anything out there listening. It's just shouting in the dark hoping someone will answer with help. There's no need to pass around instances of the Presenter, so wiring MVP triads together is a little bit simpler. Let's take a look. The first step is to define the proper events on the View interface. Here's an example from the Shipping Screen first described in the Supervising Controller post.

```
    public delegate void VoidHandler();

    public interface IShippingScreen
    {
        event VoidHandler ShippingOptionChanged;
        string ShippingOption { get;}
    }
```

The ShippingOptionChanged event is the interesting thing here. When this event fires, the Presenter needs to recalculate the cost of the selected shipment and decide whether or not other shipment options for purchasing insurance or requiring a signature are still valid. The actual View implementation is to simply relay the screen event with no interpretation:

```
        public ShippingScreen()
        {
            InitializeComponent();
            shippingOptionField.SelectedIndexChanged += new EventHandler(
                shippingOptionField_SelectedIndexChanged);
        }

        void shippingOptionField_SelectedIndexChanged(object sender, EventArgs e)
        {
            if (ShippingOptionChanged != null)
            {
                ShippingOptionChanged();
            }
        }
```

Now, on the Presenter side we have to register the event handler. Since the event has a no argument signature, all we have to do is basically tell the screen which method on the Presenter to call. No ugly anonymous delegates or ugly new SomethingDelegate(blah, blah) syntax.

```
    public class ShippingPresenter
    {
        private readonly IShippingScreen _screen;

        public ShippingPresenter(IShippingScreen screen)
        {
            _screen = screen;
            // Attach to the event in the constructor
            _screen.ShippingOptionChanged += this.ShippingOptionChanged;
        }

        public void ShippingOptionChanged()
        {
            // fetch the state the Presenter needs inside this operation
            // from the Vjiew
            string option = _screen.ShippingOption;
            // do whatever it is that you do to respond to this
            // event
        }
    }
```

And yes, I would definitely make the event handler methods public. I think you should do at least one unit test to verify that you're wired to the correct event handler on the View interface, but otherwise I think you'll find it much more convenient to test the ShippingOptionChanged() method by calling straight into the ShippingOptionChanged() method. I know the semantics of the unit test are simpler when you do it that way.

Assuming that you are going to unit test by simulating the event being raised, your unit test for the functionality above will look something like this (assuming that you use RhinoMocks anyway):

```
        [Test]
        public void HandleTheShippingOptionChangedEvent()
        {
            MockRepository mocks = new MockRepository();
            IShippingScreen screen = mocks.CreateMock<IShippingScreen>();
            // Keep a reference to the IEventRaiser for ShippingOptionChanged
            screen.ShippingOptionChanged += null;
            IEventRaiser eventRaiser = LastCall.IgnoreArguments().GetEventRaiser();
            // other expectations here for the Presenter interacting with both
            // the backend and the View
            mocks.ReplayAll();
            // Raise the event
            ShippingPresenter presenter = new ShippingPresenter(screen);
            eventRaiser.Raise();
            // Check the interactions
            mocks.VerifyAll();
        }
```

The thing to note is how we grab onto the event with RhinoMocks to simulate the event. In a normal test harness I usually build the presenter under test and all of the mock objects in the SetUp() method. That requires a little bit of knowledge about RhinoMocks to keep the tests running correctly. Check out the links at the bottom to Phil Haack and Jean-Paul for more information on how to do this. **Direct Communication**

Why do I prefer direct communication? Two reasons.

“CTRL-B” and “CTRL-ALT-B.” If the this isn’t ringing a bell, these are the ReSharper keyboard shortcuts you use inside one method to quickly navigate to another method (or class or interface or delegate, etc.). I think it’s easier to understand the codebase when there’s a direct semantic link from View to Presenter and vice versa. With direct communication I think the code is more intention revealing. When I look at the View code I know more about what happens when a dropdown value changes. With events there’s more mechanical steps to figure out what’s registered for the event. I think the mechanics for mocking events are awkward (I don’t particularly have a better suggestion than what RhinoMocks does now). Heck, I think events in general are awkward and I’ve never liked them. I would rather just treat my Presenter like an Observer of the View so I can make all of the event notification registration in a single line of code in the Presenter.

So the obvious downside is that now the View has to know about the Presenter, which spawns a new design question – who begat’s who? The majority of the time I say that it’s a Presenter-centric world. I create the Presenter first with the View coming in as a constructor argument.

```
public class ShippingScreenPresenter : IShippingScreenObserver
{
    private readonly IShippingScreen _screen;
    // The view is injected in through the constructor
    public ShippingScreenPresenter(IShippingScreen screen)
    {
        _screen = screen;
    }

    // I suppose you could do this operation in the constructor instead, but
    // it always feels so wrong to do much more than set fields in a constructor

    public void Start()
    {
        // bootstrap the screen
        _screen.AttachPresenter(this);
    }

    public void ShippingOptionChanged()
    {
        // do what you do so well
    }
}
```

I usually have some sort of explicit call to a Start() method on my Presenter's to bootstrap the screen. In this case, the very first operation in Start() is telling the View about the Presenter. I know what you're thinking, this is going to tightly couple my View to the Presenter. What if I want to use a different Presenter with the same View (it's not that uncommon)? What if I want to test the View itself? The easy answer is to treat the Presenter as an Observer of the View. In my DevTeach talk I brought up the tight coupling issue and promptly did an "Extract Interface" refactoring with ReSharper to show how easy it was to decouple the View from the concrete Presenter. No one was impressed. The View only sees:

```
public interface IShippingScreenObserver
{
    void ShippingOptionChanged();
    // all other callback methods for screen events
}
```

The event handlers in the View itself get reduced to an immediate call to the Presenter (in bold).

```
public partial class ShippingScreen : Form, IShippingScreen
{
    private IShippingScreenObserver _presenter;

    public ShippingScreen()
    {
        InitializeComponent();
        shippingOptionField.SelectedIndexChanged += new EventHandler(
            shippingOptionField_SelectedIndexChanged);
    }

    void shippingOptionField_SelectedIndexChanged(object sender, EventArgs e)
    {
        _presenter.ShippingOptionChanged();
    }

    public void AttachPresenter(IShippingScreenObserver presenter)
    {
        _presenter = presenter;
    }
}
```

In the section on MicroController, I'll show how my project is creating the screen behavior for calls to the Presenter.

Best Practices in either case

The single best advice I think I could give is to make the communication as simple as possible. As much as possible, make the callbacks and event handlers to the Presenter take in zero arguments. Make the Presenter go back to the View and/or the Model to get current state. On one had it makes the View simpler by eliminating code from the marginally testable View, on the other hand it decouples the View from needing to "Know" what the Presenter needs for its behavior determinations. The exception cases would be for things like double clicking a single line inside of a grid to launch a new screen. In that case I think the simplest thing to do is to send over just enough information in the handler for the Presenter to know which data member was the subject of the event.

The other advice I'll give is to make the method names for event handling on the Presenter be as descriptive as possible. Ideally, you should be able to almost entirely comprehend the behavior of a screen from looking at the Presenter alone.

I'll talk about integration testing in later posts in this series.

8 Answering some questions

I received some questions today that I thought were probably best addressed in a separate post. If I've confused one person, it's likely I've confused many. Here we go...

First Question

"So does the presenter provide the data from the model to the view, or does the view also have a reference to the model? Or does it depend on which of the patterns from earlier in the series that you pick?"

Short answer: Yes, sometimes, and it depends.

Long answer: By and large I believe in the primacy of the Presenter, meaning that things start with the Presenter telling the View what to do rather than the other way around. More specifically the Presenter "knows" about the Model first and then populates the View with the data in the Model. Whether or not the Presenter fetches the Model, or is handed the Model, or creates the Model is a subject I'm going to put off until the posts on creating the ApplicationShell and screen coordination. Back to the original question, how the View gets the data depends on the pattern used for the screen.

- Autonomous View — There is no separate Presenter or Controller. The View either creates the Model from scratch or gets the Model from another class

- Supervising Controller — The Presenter/Controller is responsible for getting the Model first and setting up the View with all of the data it needs, including the Model. With Supervising Controller we're still content to let the View know how to display the various properties of the Model through data binding or another mechanism. In this pattern the View definitely has a reference to the Model and is largely responsible for synchronizing screen state with the Model directly.

- Passive View — In this case the View has no reference whatsoever to the Model. The Presenter synchronizes the screen state with the Model (and vice versa) by explicitly mapping properties of the Model to public properties of the View which in turn set screen state. What you'll see in the View is a lot of getter and setter methods that delegate to screen elements like this: get return nameTextbox.Text; and set nameTextbox.Text = value;.

- Presentation Model — In this case the Presenter *is* the Model. The View binds directly to the Presentation Model just like it would in the Supervising Controller

Second Question

"I'm still wrapping my head around the Presenter. Is it tightly coupled to the view? For instance if I have some code that says turn these fields blue when the user does x, does that code live in the presenter or the view?"

That's a really good question because it skirts on some potentially treacherous ground. How much should the Presenter know about the View? How much should the View know about the Presenter? Can I swap out the View? If you read the scenario described above I'd say that you have two distinct responsibilities:

- Based on a certain user action, deciding that a visual indication should change on the screen (turning the fields blue)

- Making the visual indication on the screen

It's very often advantageous to separate out the responsibility for deciding to take an action from the responsibility of carrying out the action. You'll see this time and time again in Object Oriented Programming, especially if you employ Test Driven Development. Turning the fields blue is definitely a View responsibility to me. This code in the Presenter _view.ColorOfFooField = 'blue' feels very wrong to me. I'd rather the Presenter do this: _view.IndicateFooIsSomeCondition(). The Presenter is supposed to be about behavior, and the second choice is much more semantically meaningful to me in terms of behavior. The View is responsible for presentation and should get to decide how some sort of screen indication is made.

To summarize I guess I'd say that the Presenter really shouldn't know the intimate details of the presentation, including color. I think calls to EnableFoo() or DisableFoo() are generally correct because that is screen behavior that's generally determined by business rules and screen logic. Ideally the Presenter should not be so tightly coupled to the View that you could not swap in a completely different View class (going from WinForms to WebForms may be too much to ask though).

Just to confuse the matter, I use the term "Embedded Controller" to refer to "user interface widget-aware" classes that can help a View do some of the presentation work.

For example, if I decide I want to make some sort of common screen indication (like turning a field that has invalid data or changed data blue) a common way across the system I might create an Embedded Controller that's reused across screens. I call it an Embedded Controller because it's completely contained within a View as just a part of the View's machinery. The most immediate example I can recall is a "GridHelper" class to help bootstrap the standard "sort, page, query, filter" functionality that's similar across screens.

My Question

"How does Acropolis effect this?"

I have no idea yet, but I've heard it's supposed to be pretty cool. I would have an idea, but I missed the Acropolis session at the MVP Summit because Bellware sent me on a wild goose chase to the wrong building.

Clear as mud? You know, as many topics have come up from this series, I think this would be a good idea for a book... ...written by someone else.

9 What's the Model?

What's the Model?

I've spent most of the series talking about the View or the Presenter, but the Model piece of the triumvirate has a role to play as well.

A couple years ago I participated in a session on design patterns for fat clients that Martin Fowler was running to collect data for his forthcoming sequel to the PEAA book. One of the topics that came up in conversations afterward was whether or not it was desirable to put the real Domain Model on the client or use a mirror version of the Domain Model that you allow to have some user interface specific functionality. In other words, is the Domain Model pattern applied to the user interface a completely different animal with different rules than the traditional POCO Domain Model in the server? We didn't come up with any kind of a consensus then, and I've never made up my mind since.

As I see it, you have four choices for your Model:

- **Domain Model Class** – Just consume the real application classes. Many times it's the simplest path to take, and leaves you with the fewest moving parts. My current and previous projects both used this approach with a fair amount of success. In my current project our Domain Model classes implement quite a few business rules that come into play during screen interactions. The downsides to consuming the real Domain Model in the View are that you're binding the View more tightly to the rest of the application than may be desirable and the possibility of polluting the Domain Model with cruft to support INotifyPropertyChanged interfaces and other User Interface needs. Part of the reason I'm perfectly happy to consume Domain

Model objects directly on my project is that our screen design doesn't require any special UI support for our custom data binding solution.

- **Presentation Model** – Even if you're largely following a Model View Presenter architecture, the Presentation Model is still useful. Think of this realistic screen scenario: the real domain objects are an aggregate structure and your View definitely needs some INotifyPropertyChanged-type goo. In this particular case a Presentation Model that wraps and hides the real domain objects from the View is desirable. The Presentation Model probably provides a flattened view of the domain aggregate to make data binding smoother while implementing much of the UI infrastructure code, allowing the domain classes to take the shape that is most appropriate for the business logic and behavior and keeping them from being polluted with UI code. I should not that using a Presentation Model **will potentially add extra work to synchronize the data** with the underlying domain objects.
- **Data Transfer Object** – Forget about behavior and just use a lump of data. This is often a result of hooking the user interface directly to the return values of a web service. In my previous application we wrote an absurd amount of mapping code to map data transfer objects to domain objects and vice versa and I think we all felt like it ended up being a huge waste (we *really* needed to isolate our client from the backend, so it was mostly justified). This time around I'm honestly quite content just to bind some of the user interface directly to the Data Transfer Objects returned from our Java web services. I feel a little bit dirty about this, but this approach is dirt simple. It does couple us a bit more to the server than I would normally like, but I'm hoping to compensate with a fairly elaborate Continuous Integration scheme that does a fully integrated build with end to end StoryTeller/Fit tests anytime either codebase changes to detect breaking changes.
- **DataSet/DataTable/DataView** – See below:

Somebody has to ask, what about DataSet's? I despise DataSet's in general, but there's no denying that sometimes the easiest way to solve a problem is to revert back to Stone Age techniques and use them. There's a memorable scene from the first Lord of the Rings movie when Gandalf is speaking to Frodo very dramatically of the One Ring – "It wants to be found." It's the same way in WinForms. The user interface widgets often **want to consume a DataSet**. The entire WinForms UI toolkit was originally wrapped around a very datacentric view of the world and sometimes you just go along with it.

Alright, it's not that bad. I will happily use a DataSet/DataTable/DataView in my user interface as the ostensible Model any time it's the easiest way to use a UI widget or when I want to take advantage of the sorting and filtering capabilities of a DataTable (I think that equation changes when Linq to Objects hits). The Data* classes aren't my real domain though, I generally convert my real classes to a DataTable just in time for display. And no, a DataSet-centric approach doesn't buy me much because as I'll show in the next section the Model has real responsibilities beyond just being a dumb bag of data. Plus the little issue that DataSet's are not interoperable and we're using web services written with Java for our backend services.

While a DataSet makes the data binding generally very simple, you're left with the usual drawbacks to a DataSet. You can't embed any real logic into the DataSet, so you have to be careful with duplication of logic. If you insist on a DataSet approach, I'd recommend giving the Table Module approach some thought as a way to centralize the related business logic for a particular set of data to avoid duplication. Personally, I think DataSet's are clumsy to use inside of automated tests in terms of test setup. A strongly-typed DataSet helps to at least get some Intellisense, but they annoy me as well. Plus you'll often find yourself building data that's meaningless to the test just to satisfy the referential integrity rules of a DataSet. That's wasted effort.

10 Assigning Responsibilities in a Model View Presenter Architecture

Where should this code go?

In a post last winter I said that a coder only becomes a true software craftsman when they start asking themselves "where should this code go?" It's a neverending question that you use to guide your designs and assign responsibilities. In a typical Model View Presenter architecture the screens are composed of 4 basic types of classes ("M", "V", "P", and the Service Layer). When you're designing a screen along MVP lines, you need to assign each responsibility of the screen to one of these 4 pieces, while also determining the design constraints upon each piece. The question of "who does what?" isn't perfectly black and white, and there are many variations on the basic structure. That being said, here's my best advice on the duties and constraints of each of the four pieces. As a rule of thumb, try to put any given responsibility as close to the top of this list as possible without violating the design constraints of each piece. **Service Layer**

The Service Layer classes are generally a Facade over the rest of the application, generally encompassing business logic and data persistence. In general it's important and valuable to decouple the view layer from the rest of the application as much as possible to create orthogonality between the backend and the user interface. If you find the Presenter collaborating with more than a single Service Layer class you might think about putting a single Facade over the two services for that screen.

In terms of constraints, the Service Layer is not aware of anything specific to any one screen, but it is perfectly permissible for the Service Layer classes to be aware of the Model and perform work on the Model. If you've chosen to use actual Domain Model classes for the Model in the screen, the Service Layer is probably just some sort of Repository class. You might think of it like this. Say you're building a monolithic application today, but tomorrow you want to expose web services to interact with the system as an alternative to the user interface. The Service Layer classes should be able to work within the web service code without any change.

The Service Layer should encompass any type of business logic that is not specific or tightly coupled to a specific screen. In my current system there is going to be a validation on certain date values against a calendar service that compares the selected date against the business days for a particular country. If I were to embed that logic into the Presenter for my first Trade screen the logic wouldn't be very accessible to the following Trade screens. That calendar logic definitely belongs in the Service Layer.

Model

The Model can be more than just a lump of data. For a multitude of reasons, I want my Model class to implement most of the business rules for the given business concept. The first reason is simply cohesion. As much as possible, I want all of my business rules for an Invoice or a Trade or a Shipment in the domain class for that particular domain concept. If I want to look up the business rules for an Invoice, I want a single place to go look. One of my primary design goals is to put code where you would expect to find it, and to me that means that invoicing business rules go in the Invoice class.

The second reason is reuse or elimination of duplication. If you make a Presenter or the View itself responsible for business rules you're much more likely to find yourself duplicating the same business rules across multiple screens. Those declarative validation controls in ASP.Net are sure easy to use, but there's a definite downside because the responsibility for validation is in the wrong place.

For example, the Model classes in my current system are responsible for:

- **Validation rules**. The validation rules for a domain concept should definitely be kept in the Domain Model. Validation logic is most definitely a business rule. Besides, it's just so common to have multiple screens acting upon the same classes. You don't really want to have to duplicate validation rules from a "Create Trade" screen to the parallel "Edit Trade" screen do you? If nothing else, putting validation rules in the Domain Layer makes these rules very simple to test compared to the same rules living in either the View or even the Presenter. I've got a lot more to say on this subject in the next post on the Notification pattern. In the meantime, Jean-Paul has a good post on Validation In The Domain Layer – Take One and again Validation In The Domain Layer – Take Two.
- **Default values**. My last two projects have included quite a bit of logic around assigning default values. The normal scenario is that the use selects one value, and from that one value you can determine logical defaults for one or more other fields. I have a little business rule to code tomorrow morning that will automatically set the values for two date fields to 2 days after the Trade Date as soon as the Trade Date is selected for the first time. That rule is going right into the appropriate Trade subclass, both because it's where I'd expect to find that code and because it makes

that business rule very easy to drive with Test Driven Development. Because the rule is wholly implemented in the Model class and the Model class is completely decoupled from the View and Service Layer, I can test these rules by applying purely state based testing. For this approach to work I do need the screen to automatically synchronize itself with the Model when the Model changes, but the WinForms tooling makes that kind of Observer Synchronization relatively simple.

- **Calculated values**. Again, this is dependent upon using Observer Synchronization, but I place the responsibility for calculating derived values into the Model class. It's a business rule, and the Model class has all of the data, so it's the natural place for this logic. As with defaulted values, the calculation can be relatively simple to test in isolation.

The constraint on the Model is that whether or not we allow the Model classes to call out to any kind of service, and if so, how much coupling do we allow with other services? In my current project I don't allow any direct coupling from my Model classes to any external system. If a business rule for defaulting or calculating a value requires information or a service from outside the Model, I would partially move that responsibility out into the Presenter to keep the Model decoupled from infrastructure. The Model classes don't know how they're displayed nor how they're persisted and updated. In other systems you might opt for more of an Active Record approach that puts some form of service communication responsibility into the Model classes.

The Presentation Model approach isn't really an exception to these constraints because the Presentation Model itself usually wraps the actual Model. **View & Presenter**

The screen itself is split between two main actors, the View and the Presenter. As we've seen in the previous posts on the Supervising Controller, Passive View, and Presentation Model, there's a couple different ways to split responsibilities between the View and Presenter. Rather than rehash those posts, I just want to add a few more thoughts:

- Don't allow the View to spill out into the Presenter. There might be exception cases, but don't allow any reference to any Type in the System.Windows.Forms namespace from the Presenter class. Any WinForms mechanics in the Presenter is like letting water splash out of the tub and rot the floor.

- Keep the View as thin as possible. Always ask yourself if a given piece of code could or should live outside of the View. My advice is to move anything out of the View that doesn't have to be there.

- Don't allow the Presenter to become too big. It's far too tempting to use the Presenter as a receptacle for any random responsibility. If you see parts of the Presenter that don't seem to be related to the rest of the Presenter, do an Extract Class refactoring to move that code to a smaller, cohesive class.

- There is no ironclad rule that says 1 screen == 1 view + 1 presenter. It's often going to be valuable to reduce the complexity of either the Presenter or View by breaking off pieces of the screen into smaller pieces. There may still be one uber-Presenter and one uber-View for the entire screen, but don't hesitate to make specific Presenter or View classes for a part of a complicated screen. There's also no ironclad law that says n views = n presenters as well.

Getting back to first causes, it all comes down to two things:

- Is each piece of the code easy to understand?
- Is the code easy to test?

Answer these two questions in the affirmative and I think your design is effective.

11 Domain Centric Validation with the Notification Pattern

Domain Centric Validation

As I said in the last post, it's highly advantageous to put the validation rules into the Model classes. Just to recap, I'll rattle off four reasons why I want my validation rules in my Model classes:

- Validation is mostly a business logic function anyway, and purely for the sake of cohesion it makes sense to put business rules into business classes. As much as possible, I want related business rules in a single place rather than scattered about my code.
- Since it's fairly common to have a single Model class edited in multiple screens, I want to reuse the validation logic consistently across these different screens. More importantly, I want to eliminate duplicated logic between these screens. It's going to be much easier to avoid duplication by placing this functionality in the Model than it would be to put it in each Presenter or View. I know somebody is going to bring up the idea of drag n'drop validator's, but even those are potentially easy to use, the potential duplication can easily become a tax — especially as your validation rules change.
- By decoupling the validation logic from the View and the backend, we've made these validation rules easy to test. The easiest possible type of unit testing is simple state based testing. Push a couple values into a single object and check the return value with no interaction with anything else. Clear sailing ahead.
- I'm going to content that It's easier to read the validation logic. The downsides of Rapid Application Development (RAD) approaches are widely known, but for me the worst part is that I think RAD designers and wizards obscure the meaning of code and camouflage the functionality in design time goo.

Ok, that's a lot of hot air about why we want to do domain centric validation, but how do I do it? My current project is doing domain centric validation, and so far I'm thrilled with how it's turning out. Here's our secret sauce (ok, it's not that secret because it's a documented design pattern):

The Notification Pattern

In doing validation on user input there are two main tasks:

- Perform the validation and create meaningful validation messages.
- Display these validation messages to the user on the screen in a meaningful way. In WinForms development we've got the handy ErrorProvider class that probably meets the needs of most scenarios.

Let me emphasize this a little bit more, we have two separate concerns: validation rules and screen presentation of validation failures. As always, we can make our development simpler by tackling one problem at a time. The advantage in testability, reuse, and understandability I mentioned above is largely predicated on separating the validation logic from the screen machinery.

Assuming you've accepted my advice to separate the two tasks and make the validation logic completely independent of the presentation machinery, we can move onto the next question. How can we marshal the validation messages to the View in a way to simplify the presentation of validation messages? In other words, how does a validation message get put into the screen at the right place? There is a ready made answer for us, just use the Notification pattern as a convenient way to package up validation messages.

From Martin Fowler, the Notification pattern is a

An object that collects together information about errors and other information in the domain layer and communicates it to the presentation.

Let's take a look at my project's Notification structure. The first piece is a class called NotificationMessage (the code below is somewhat elided) that's simply a single validation error consisting of the message itself and the name of the Property on the Model that the message applies to:

```
    public class NotificationMessage : IComparable
    {
        private string _fieldName;
        private string _message;

        public NotificationMessage(string fieldName, string message)
        {
            _fieldName = fieldName;
            _message = message;
        }

        public string FieldName
        {
            get { return _fieldName; }
            set { _fieldName = value; }
        }

        public string Message
        {
            get { return _message; }
            set { _message = value; }
        }

        // Override the Equals method to make declarative testing easy
        public override bool Equals(object obj)
        {
            if (this == obj) return true;
            NotificationMessage notificationMessage = obj as NotificationMessage;

            if (notificationMessage == null) return false;
            return Equals(_fieldName, notificationMessage._fieldName) && Equals(_message, notificationMessage
                ._message);
        }

        // Override the ToString() method to create more descriptive messages within
        // xUnit testing assertions
        public override string ToString()
        {
            return string.Format"(Field {0}: " {1}, _fieldName, _message);
        }
    }
```

For reasons that will be clear in the section on consuming a Notification, it's very useful to tie the messages directly to the property names. The second part is the Notification class itself. It's not much more than a collection of NotificationMessage objects:

```
    public class Notification
    {
        public static readonly string REQUIRED_FIELD = "Required "Field;
        public static readonly string INVALID_FORMAT = "Invalid "Format;
        private List<NotificationMessage> _list = new List<NotificationMessage>();

        public bool IsValid()
        {
            return _list.Count == 0;
        }

        public void RegisterMessage(string fieldName, string message)
        {
            _list.Add(new NotificationMessage(fieldName, message));
        }

        public string[] GetMessages(string fieldName)
        {
            List<NotificationMessage> messages = _list.FindAll(delegate(NotificationMessage m)
      { return m.FieldName == fieldName; }
      );

            string[] returnValue = new string[messages.Count];
            for (int i = 0; i < messages.Count; i++)
            {
                returnValue[i] = messages[i].Message;
            }

            return returnValue;
        }

        public NotificationMessage[] AllMessages
        {
            get
            {
                _list.Sort();
                return _list.ToArray();
            }
        }

        public bool HasMessage(string fieldName, string messageText)
        {
            NotificationMessage message = new NotificationMessage(fieldName, messageText);
            return _list.Contains(message);
        }
    }
```

Easy enough right? You've got everything you need to do domain centric validation — except for all the stuff around the Notification class that does all the real work. Details. Or as my Dad's construction crew will say after day one on a new project, "all we lack is finishing up."

Filling up the Notification

The first design goal is simply to come up with a way for a Model object to create Notification objects. Most validation rules are very simple and take one of a handful of forms, so it's worth our while to make the implementation of these simple rules as quick and declarative as possible. We also need to relate any and all validation messages to the proper field. Since this is .Net, the easiest usage scenario is simple attributes that we can use to decorate property declarations. Using attributes comes with the great advantage of quickly tying a validation rule to a particular property.

Using attributes always requires some sort of bootstrapping code that finds and acts on the attributes decorating code, but I'm going follow my own "Test Small Before Testing Big" rule. Let's put off the overall validation calculation to tackle the very small goal of creating validation attributes. Once we have the shape of the attributes and maybe the Model class down, the validation coordination code largely falls out.

Don't use the attributes as just markers though. The actual validation logic should reside in the attribute itself to allow for easier expansion of our validation rules. This is a good example of the Open/Closed Principle. We can create all new validation logic by implementing a whole new ValidationAttribute without having to modify any existing code. That's good stuff. I think this is also an example of Craig Larman's Protected Variations concept.

All of our validation attributes inherit from the ValidationAttribute shown below:

```
    [AttributeUsage(AttributeTargets.Property)]
    public abstract class ValidationAttribute : Attribute
    {
        private PropertyInfo _property;

        public void Validate(object target, Notification notification)
        {
            object rawValue = _property.GetValue(target, null);
            validate(target, rawValue, notification);
        }

        // Helper method to write validation messages to a Notification
        // object with the correct Property name

        protected void logMessage(Notification notification, string message)
        {
            notification.RegisterMessage(Property.Name, message);
        }

        // A Template Method for the actual validation logic
        protected abstract void validate(object target, object rawValue, Notification notification);

        // The Property of the targeted class
        // that's being validated

        public PropertyInfo Property
        {
            get { return _property; }
            set { _property = value; }
        }

        public string PropertyName
        {
            get
            {
                return _property.Name;
            }
        }
    }
```

Attributes don't find themselves, so we'll need to "push" the correct actual System.Reflection.PropertyInfo object to the ValidationAttribute objects, so there's an open getter/setter for the PropertyInfo. The validation will work by first finding all of the special validation attributes, attaching the proper PropertyInfo's, and looping through each ValidationAttribute and calling the Validate(target, notification) method. All the ValidationAttribute classes do is register messages on a Notification object, so there's absolutely no coupling to either the server or the user interface.

To actually build a specific ValidationAttribute you simply inherit from ValidationAttribute and override the validate(target, rawValue, notification) Template Method. The simplest case for a simple required field validation is shown below:

```
[AttributeUsage(AttributeTargets.Property)]
public class RequiredAttribute : ValidationAttribute
{
    protected override void validate(object target, object rawValue, Notification notification)
    {
        if (rawValue == null)
        {
            logMessage(notification, Notification.REQUIRED_FIELD);
        }
    }
}
```

In real life you'd have to enhance this a bit to handle primitive types that aren't nullable, but right now this is doing everything my project needs. Using this little monster is as simple as marking a Property like this;

```
    [Required]
    public string Name
    {
        get { return _name; }
        set { _name = value; }
    }
```

or combine attributes:

```
    [Required, GreaterThanZero]
    public double? BaseAmount
    {
        get { return _baseAmount; }
        set { _baseAmount = value; }
    }
```

Of course, some rules are going to be too complex or simply too rare to justify building an attribute (think about rules that only apply in a certain state or involve many fields). To allow for these more complex cases, I chose to use a simple marker interface called IValidated.

```
public interface IValidated
{
    void Validate(Notification notification);
}
```

A Model class can implement this interface as a kind of hook to perform any sort of complex validation that just doesn't fit into the attribute-based validation. There's a bit of a design point I'd like to make here. This little IValidated interface is an example of the Tell, Don't Ask principle in action. Rather than querying the attributes and the validated object to decide what rules to enforce, we can create far more variations and allow for far more extension by just telling the object being validated and each attribute class to add it's messages to the Notification object.

A quick side note. If you're going to build something like this for yourself, I'd recommend building the first couple specific cases, then doing an Extract Superclass refactoring to "lift" the common template into a superclass. Starting by designing the abstract class can sometimes lead into a black hole. When I built this code I created the RequiredFieldAttribute first then used ReSharper to create the Supertype.

And yes, I do know about the Validation Application Block, but it's kind of fun and fairly easy to write our own to do exactly what we want. Besides, I'm not sure how well the Validation Block plays in the Notification scenario. My real inspiration here is the Notification-style validation baked into ActiveRecord in Ruby on Rails. The same kind of thing that we did with attributes in C# is potentially more declarative in a Rails model. Just coincidentally, Steve Eichert has a cool writeup on this subject at Create DDD style specifications in Ruby with ActiveSpec.

Putting the Notification Together

We've got the Notification class, some working ValidationAttribute classes, and the IValidated hook interface. Now we need a Controller class of some kind to glue all of these pieces together. With my nearly infinite creativity, I've named this class in our system "Validator." The next task is simply to scan a Type and create a list of ValidationAttribute objects for each declared attribute. That's done with this static method:

```

        // Find all the ValidationAttribute's for a given Type, and connect
        // each ValidationAttribute to the correct Property

        public static List<ValidationAttribute> FindAttributes(Type type)
        {
            List<ValidationAttribute> atts = new List<ValidationAttribute>();

            foreach (PropertyInfo property in type.GetProperties())
            {
                Attribute[] attributes = Attribute.GetCustomAttributes(property, typeof (ValidationAttribute)
                    );

                foreach (ValidationAttribute attribute in attributes)
                {
                    attribute.Property = property;
                    atts.Add(attribute);
                }
            }
            return atts;
        }
```

Once that method is unit tested, we can move onto the core task of validating an object. That's accomplished with the method below.

```
        public static Notification ValidateObject(object target)
        {
            // Find all the ValidationAttribute's attached to the properties
            // of the target Type

            List<ValidationAttribute> atts = scanType(target.GetType());
            Notification notification = new Notification();

            // Iterate through each ValidationAttribute and give each
            // a chance to add validation messages
            foreach (ValidationAttribute att in atts)
            {
                att.Validate(target, notification);
            }

            // And if the target implements IValidated, call the
            // Validate(Notification) method to do the special cases

            if (target is IValidated)
            {
                ((IValidated)target).Validate(notification);
            }

            return notification;
        }
```

I've elided the code here for brevity, but the first thing ValidateObject(object) does is to call its scanType(Type) method to find the list of ValidationAttribute's declared for target.GetType(). Since Reflection isn't the fastest operation in the world, I do cache the list of ValidationAttribute objects for each Type on the first access for that Type.

Once we have the ValidationAttribute list everything else just falls out. **Testing Validation Logic**

One of the best things about this approach is how easy automated testing of the validation logic becomes. Just create the class under test, pump in some property values, create a Notification from its current state, and finally compare the Notification object to the expected results. One of Jeremy's Laws of TDD is to "go declarative whenever possible" in testing. We're using Fit tests[LINK] to perform declarative testing of our validation rules, and this is exactly the kind of scenario where Fit shines. I will often embed bits of Fit tests directly into NUnit tests for developer tests, especially for set based functionality. This might not be very interesting to you unless you already use the FitnesseDotNet engine, but we create and define the state of one of our Model objects inside a Fit table, then turn around and check the validation messages produced for the input. The Fit test itself looks something like this:

!|UserFixture|
|Create New|
Name	Jeremy
Birthday	BLANK
PhoneNumber	BLANK
The Validation Messages Are
FieldName	Message
Name	Field Required
Birthday	Field Required

The actual UserFixture class would inherit from DoFixture and provide methods to set each public property (we codegen these Fixture classes from reflecting over the targeted classes, i.e. the code generator creates a UserFixture class for User). Once the desired state of the object is configured, the validation logic is tested by comparing the actual validation messages against the expected messages. At the bottom of the Fit test above we simply make a declaration that the expected validation messages are the values in the nested table (the bolded, red text). The Fit engine has a facility called the RowFixture that makes this type of set comparison very easy. The RowFixture to check the validation messages is triggered by this method on the UserFixture class:

```
        public Fixture TheValidationMessagesForFieldAre(string fieldName)
        {
            Notification notification = Validator.ValidateObject(CurrentTrade);
            string[] messages = notification.GetMessages(fieldName);
            return new StringArrayFixture(messages);
        }
```

This section might not be all that useful to you if you're not familiar with the mechanics of Fit testing, but then again, you might want to go give Fit another try. It's goofy alright, but it's very useful in the right spot. **Consuming Notifications**

Alrighty then! We've got ourselves a Notification object that relates all the user input failure messages to the relevant properties of the Model. Now we've just got to get these messages to the right place on the screen. In your best Church Lady voice — who knows about both the field names and the controls that edit those fields? Could it be.... ...Satan! The data binding! I'm not going to show it here because I suspect the code would be nasty, but you could do a recursive search through all the child Control's and check the DataBindings collection of each child control to see if it's bound to any property of the object being validated.

Alternatively, you ditch the designer for data binding and put just a small Facade over the BindingDataSource class. If you build just a tiny class that wraps BindingDataSource with something like "BindControlToProperty(control, propertyName)" you could capture a Dictionary of the controls by field name along that way that could make attaching the errors much simpler. One of the themes I was trying to get across with my WinForms patterns talk at DevTeach was that sometimes a codecentric solution can be much more efficient than the equivalent work with designers.

Or, if that sounds as nasty to you as it does to me, you could just write your own synchronization mechanism that bakes in the functionality to attach validation messages to controls. If you thought I was nuts to suggest that mere mortals could write their own CAB analogue, you'll really flip out once I admit that I've written a replacement for WinForms data binding to do screen synchronization (so far the Crackpot idea is working out fairly well). I'll give it a much better treatment in the post on MicroController, but for now, here's a sneak peek at the code we use to attach validation messages to the proper screen element with an ErrorProvider.

```
        public void ShowErrorMessages(Notification notification)
        {
            foreach (IScreenElement element in _elements)
            {
                string[] messages = notification.GetMessages(element.FieldName);
                element.SetErrors(messages);
            }
        }
```

Each IScreenElement is what I call a "MicroController" that governs the behavior and binding of a single control element. Each IScreenElement already "knows" the Error-Provider for the form or control by this point, so it just needs to turn around and use the ErrorProvider to attach messages to its internal control.

12 Unit Testing the UI with NUnitForms

I've frequently read critiques of Test Driven Development that question it's effectiveness or practicality in regards to user interface code. It's certainly more challenging to test drive user interface code, but it’s still very practical. There is some specific “lore” that TDD practitioners have created over the last couple years to address automated unit testing against user interface code. The best single answer in my book is to utilize Humble View’s of some sort or another to remove as much code as possible from the clutches of the WinForms presentation infrastructure. At some point though, you’re left with the actual View code and it’s a nice cozy home for bugs to breed in. You can take a rationalized approach and say that you’ve reduced the View to code that’s so simple that it’s not worth the time investment to unit test the View’s with automated tests. Another option that we’ll explore here is to unit test the WinForms View classes with the NUnitForms library. **NUnitForms**

I’ve been meaning to write a post on using NUnitForms for a long time because it’s a great tool that’s never gotten the attention that I think it deserves. NUnitForms is a library that can be used from within any of the xUnit tools (NUnit/MbUnit/MSTest) to enable unit testing of View behavior. NUnitForms can drive a screen by finding and manipulating the controls on a screen, as well as check properties of controls at runtime. For the most part I only use NUnitForms to write very simple unit tests on the wiring from View to Presenter, but NUnitForms can go much farther than that. In an earlier post I decried the usage of the Autonomous View as a generally untestable and unmaintainable mess. While I think that’s definitely the case for a screen of any nontrivial complexity, we can happily unit happily forgo View/Presenter separation for smaller screens without sacrificing unit test coverage. Let’s take one last look at the sample Shipping screen from the Supervising Controller post and see how we could test that screen with NUnitForms if we had taken an autonomous view approach.

Here’s part of the code for that screen:

```
public partial class ShippingScreen : Form
{
    private IShippingService _service;
    private Shipment _shipment;

    public ShippingScreen()
    {
        InitializeComponent();

        // Grab an instance of IShippingService from StructureMap
        _service = ObjectFactory.GetInstance<IShippingService>();
        // set the Shipment property and start up the data binding
        Shipment = new Shipment();
        shippingVendorField.SelectedIndexChanged += new System.EventHandler(
            shippingVendorField_SelectedIndexChanged);
        shippingOptionField.SelectedIndexChanged += new System.EventHandler(
            shippingOptionField_SelectedIndexChanged);
    }

    // Reset the checkbox enabled states anytime a different Shipping Option is selected
    void shippingOptionField_SelectedIndexChanged(object sender, System.EventArgs e)
    {
        resetDeliveryOptions();
    }

    // When a new vendor is selected, fill the Shipping Option ComboBox
    // and reset the Delivery Option checkbox's
    void shippingVendorField_SelectedIndexChanged(object sender, System.EventArgs e)
    {
        // When the Vendor is selected, cascade the list for Shipping Options
        string[] options = _service.GetShippingOptions(_shipment);
        shippingOptionField.Items.Clear();
        shippingOptionField.Items.AddRange(options);
        shippingOptionField.SelectedIndex = 0;
        resetDeliveryOptions();
    }

    private void resetDeliveryOptions()
    {
        DeliveryOptions options = _service.GetDeliveryOptions(_shipment);
        purchaseInsuranceField.Enabled = options.PurchaseInsuranceEnabled;
        requireSignatureField.Enabled = options.RequireSignatureEnabled;
    }

    public Shipment Shipment
    {
        set
        {
            // start up the data binding stuff
            _shipment = value;
        }
    }
}
```

Easy enough. Let's start the unit test harness. We'll create our normal NUnit TestFixture class, but this time we'll inherit from NUnitForm's NUnitFormTest class to utilize some of it's bookkeeping code.

```
    // You don't have to inherit from NUnitFormTest, but it adds some
    // bookkeeping functionality that you may want
    [TestFixture]
    public class ShippingScreenTester : NUnitFormTest
    {
        private ShippingScreen _screen;
        private ComboBoxTester stateComboBox;
        private ComboBoxTester shippingVendorComboBox;
        private CheckBoxTester purchaseInsuranceCheckBox;
        private CheckBoxTester requireSignatureCheckBox;
        private TextBoxTester costTextBox;
        private ComboBoxTester shippingOptionComboBox;
        private MockRepository _mocks;
        private IShippingService _service;

        // Override Setup()
        public override void Setup()
        {
            // I'm using a mock object for the ShippingService here
            // I could just as easily have used a stub, but I can't stand
            // having those cluttering up the code.
            // It's somewhat personal preference
            _mocks = new MockRepository();
            _service = _mocks.CreateMock<IShippingService>();
            ObjectFactory.InjectStub(_service);
            _screen = new ShippingScreen();
            _screen.Show();

            // Create a ControlTester for each field on the screen for later
            stateComboBox = new ComboBoxTester"("stateField);
            shippingVendorComboBox = new ComboBoxTester"("shippingVendorField);
            purchaseInsuranceCheckBox = new CheckBoxTester"("purchaseInsuranceField);
            requireSignatureCheckBox = new CheckBoxTester"("requireSignatureField);
            costTextBox = new TextBoxTester"("costField);
            shippingVendorComboBox = new ComboBoxTester"("shippingOptionField);
        }

        // VERY IMPORTANT!  CLOSE THE SCREEN WHEN YOU'RE DONE!
        public override void TearDown()
        {
            _screen.Dispose();
        }
    }
```

Take a long look at the Setup() method because there's a lot going on. The first 3 lines are just setting up the RhinoMocks for IShippingService and directing StructureMap to return the mock object whenever an instance of IShippingService is requested (that's what the ObjectFactory.InjectStub<T>(T value) method is doing). The next two lines just create a new instance of the ShippingScreen form and make it visible. That's all you need to do. NUnitForms will happily latch onto the visible form later.

The next thing is to set up "ControlTester" objects for each of the UI controls on the screen. Each ControlTester is responsible for finding the named control and provides convenience methods to both manipulate a control and check it's properties. There are prebuilt ControlTester classes for all of the most common controls plus a generic ControlTester that can be used for everything else. You can also use a GenericTester like this one:

```
        [Test]
        public void EnableAndDisableCheckboxsWhenTheShippingOptionIsChanged()
        {
            // First, document some preconditions to prove that the code actually did something
            Assert.IsFalse(requireSignatureCheckBox.Properties.Enabled);
            Assert.IsFalse(purchaseInsuranceCheckBox.Properties.Enabled);

            string theShippingOptionSelected = "Next "Day;  // I think I picked up the habit of naming test
                data
                                            // variables ""theSomething from Colin Kershaw, but
                                            // I don't completely remember.

            DeliveryOptions theDeliveryOptionsReturnedFromTheService = new DeliveryOptions();
            theDeliveryOptionsReturnedFromTheService.PurchaseInsuranceEnabled = true;
            theDeliveryOptionsReturnedFromTheService.RequireSignatureEnabled = true;

            // Look at the Constraint.  Little RhinoMocks trick to verify that the ""ShippingOption
            // property on the internal Shipment object has been set prior to making the call to
            // IShippingService.GetDeliveryOptions(Shipment)
            Expect.Call(_service.GetDeliveryOptions(null))
                .Return(theDeliveryOptionsReturnedFromTheService)
                .Constraints(new PropertyIs"("ShippingOption, theShippingOptionSelected));

            _mocks.ReplayAll();

            // Select a Shipping Option
            shippingOptionComboBox.Enter(theShippingOptionSelected);

            _mocks.VerifyAll();

            // Check the enabled property on the two checkbox's
            Assert.AreEqual(
                theDeliveryOptionsReturnedFromTheService.PurchaseInsuranceEnabled,
                purchaseInsuranceCheckBox.Properties.Enabled);

            Assert.AreEqual(
                    theDeliveryOptionsReturnedFromTheService.RequireSignatureEnabled,
                    requireSignatureCheckBox.Properties.Enabled);
        }
```

Back to our shipping screen. Here's the unit test for the cascading dropdown logic when a state or province is selected. The unit test below does the following:

Set up an expectation on the mock object for IShippingService to return an array of shipping vendor names Select an option on the dropdown box for “State or Province” (the call to stateComboBox.Enter()) Verify the mock object interaction with the IShippingService Lastly, check the values on the dropdown box for the shipping vendor

```
        [Test]
        public void SelectingAStateOrProvinceCascadesTheSelectionListOfVendors()
        {
            // Setup the IShippingService to return a string array of vendors for a given State
            string[] theOptions = new string"[]{Vendor "A, "Vendor "B, "Vendor "C};
            string theStateOrProvince = ""TX;

            Expect.Call(_service.GetShippingVendorsForLocation(theStateOrProvince))
                .Return(theOptions)
                .Constraints(new Equal(theStateOrProvince));

            _mocks.ReplayAll();
            stateComboBox.Enter(theStateOrProvince);
            _mocks.VerifyAll();

            // Lastly, let's check the dropdown list on shippingVendor ComboBox
            // The ""Properties property is a reference to the System.Windows.Forms.ComboBox
            // object.  You can write assertions against anything on its public API
            Assert.AreEqual(theOptions, shippingVendorComboBox.Properties.Items);
        }
```

Now, let's look at a unit test for enabling/disabling the insurance and signature required checkbox's when the Shipping Option is selected:

```
        [Test]
        public void EnableAndDisableCheckboxsWhenTheShippingOptionIsChanged()
        {
            // First, document some preconditions to prove that the code actually did something
            Assert.IsFalse(requireSignatureCheckBox.Properties.Enabled);
            Assert.IsFalse(purchaseInsuranceCheckBox.Properties.Enabled);

            string theShippingOptionSelected = "Next "Day;  // I think I picked up the habit of naming test
                data
                                                // variables ""theSomething from Colin Kershaw, but
                                                // I don't completely remember.

            DeliveryOptions theDeliveryOptionsReturnedFromTheService = new DeliveryOptions();
            theDeliveryOptionsReturnedFromTheService.PurchaseInsuranceEnabled = true;
            theDeliveryOptionsReturnedFromTheService.RequireSignatureEnabled = true;

            // Look at the Constraint.  Little RhinoMocks trick to verify that the ""ShippingOption
            // property on the internal Shipment object has been set prior to making the call to
            // IShippingService.GetDeliveryOptions(Shipment)

            Expect.Call(_service.GetDeliveryOptions(null))
                .Return(theDeliveryOptionsReturnedFromTheService)
                .Constraints(new PropertyIs"("ShippingOption, theShippingOptionSelected));

            _mocks.ReplayAll();

            // Select a Shipping Option
            shippingOptionComboBox.Enter(theShippingOptionSelected);
            _mocks.VerifyAll();
            // Check the enabled property on the two checkbox's
            Assert.AreEqual(
                theDeliveryOptionsReturnedFromTheService.PurchaseInsuranceEnabled,
                purchaseInsuranceCheckBox.Properties.Enabled);

            Assert.AreEqual(
                    theDeliveryOptionsReturnedFromTheService.RequireSignatureEnabled,
                    requireSignatureCheckBox.Properties.Enabled);
        }
```

Testing View to Controller Communication

Most of our NUnitForms unit tests are just verifying the wiring of a View to a Presenter. As expected, I've gotten a mostly negative reaction to my preference for making the View communicate directly with the Presenter. One of the commenters questioned whether or not my direct communication would lead to harder testing. Here's a sample of testing both direct communication and communication through events. I don't think you're going to see much difference in effort either way.

Here's the scenario: when the "Save" button is clicked, call the Save() method on the Presenter.

```
public interface IPresenter
{
    void Save();
}

public class SomeScreen : Form
{
    private IPresenter _presenter;
    private Button saveButton;

    public void AttachPresenter(IPresenter presenter)
    {
        _presenter = presenter;
    }
}

[TestFixture]
public class DirectCommunicationTester
{
    private SomeScreen _screen;
    private MockRepository _mocks;
    private IPresenter _presenter;

    [SetUp]
    public void SetUp()
    {
        _screen = new SomeScreen();
        // Setup RhinoMocks for the IPresenter and attach it
        _mocks = new MockRepository();
        _presenter = _mocks.CreateMock<IPresenter>();
        _screen.AttachPresenter(_presenter);
        _screen.Show();
    }

    [TearDown]
    public void TearDown()
    {
        _screen.Dispose();
    }
    public delegate void VoidHandler();
    // A helper function.  This pays off over the application
    private void assertClickingButtonCalls(string buttonName, VoidHandler handler)
    {
        // Set up the expectation on the Presenter
        handler();
        _mocks.ReplayAll();
        // Find the button on the screen
        ButtonTester button = new ButtonTester(buttonName);
        // Click the button
        button.Click();
        // verify the interaction with the Presenter
        _mocks.VerifyAll();
    }

    [Test]
    public void ClickTheSaveButtonCallsSaveOnThePresenter()
    {
        assertClickingButtonCalls"("saveButton, delegate {_presenter.Save();});
    }
}
```

It's important to note that you generally make all calls to the presenter take in zero arguments. We'll see that same pattern repeat for event communication. You can go farther with the test helper methods to make testing even more declarative.

Now, the same thing with event driven communication using an anonymous delegate as the event handler:

```
    public delegate void VoidHandler();

    public class EventScreen : Form
    {
        private IPresenter _presenter;
        private Button saveButton;
        public event VoidHandler OnSave;
    }

    [TestFixture]
    public class EventCommunicationTester
    {
        private EventScreen _screen;

        [SetUp]
        public void SetUp()
        {
            _screen = new EventScreen();
            _screen.Show();
        }

        [TearDown]
        public void TearDown()
        {
            _screen.Dispose();
        }

        [Test]
        public void ClickTheSaveButton()
        {
            bool IWasCalled = false;
            _screen.OnSave += delegate { IWasCalled = true; };
            Assert.IsTrue(IWasCalled);
        }
```

So it's okay to use Autonomous View after all?

Because of NUnitForms I'm perfectly happy to use the autonomous view style on simple dialog screens, but that's just about it. There's a certain fuzzy point where I'd definitely choose to go to a separated presentation as opposed to an autonomous view. Even though the shipping screen above is small, I think it already crosses that line. Many people will term any kind of Model View Presenter (MVP) architecture as more complicated than the autonomous view because of the additional pieces, but I see it differently. I reject the notion that more classes automatically equals more complexity. I'd much rather have more classes if that enables me to keep any one single class as simple as possible. Testability is a huge part of maintainability in my book, but separation of concerns by itself will make the code better organized and easier to understand.

At least in the .Net world the MVP style of construction isn't being received well by mainstream developers because it's a foreign approach that isn't encouraged by the .Net tooling. In a way MVP is more complex in .Net just because the tooling doesn't lead you there. I wouldn't say that you have to fight the .Net tooling to do separated presentation, but you've got to make all of the decisions yourself.I guess it's easy to understand the negative reaction I see to MVP because it's not part of the .Net canon yet. I haven't taken a hard look at Acropolis yet, but it apparently encourages some sort of View/Presenter separation. That's a positive change in direction in .Net tooling that'll do more good than a thousand blog posts on MVP.

Lessons Learned

NUnitForms is definitely meant for unit testing, and that's where I've found it to be most applicable. On my last project we tried to drive the WinForms application in FitNesse acceptance tests with NUnitForms and struggled mightily. The complexity of the testing effort goes up dramatically when you move from testing one form or control to the full application at one time. I'm using StoryTeller/FitnesseDotNet for story testing of the user interface quite successfully on my current project, but that's a later post.

The Future

NUnitForms was written by Luke Maxon with some contributions from Levi Haskell (probably the best developer I've ever worked with). Since Luke's off living the exciting start up life and coding in Java, I've volunteered(?) to be one of the SourceForge administrators for the NUnitForms project now. I'm thinking of making some additions to NUnitForms for better support of acceptance testing and possibly some prebuilt Fit fixtures, but I'm not sure of either direction or timing. Since my company does so much WinForms work, I'm going to try to rook some of my colleagues into helping too. I'm also kicking around the idea of creating a sort of testing DSL with NUnitForms to support something reminiscent of Brian Marick's wireframe test idea, and possibly use that for acceptance testing as opposed to Fit tests. I definitely want to add some custom assertions and more options for locating controls. Got any ideas or requests? Here's some loose conceptual code, feel free to let rip with what you think:

```
using (NUnitFormsSpecification specification = new NUnitFormsSpecification)
{
  Click.Button.WithText("Do Something");
  Click.Button.Named("saveButton");
  CheckBox.WithText("Some flag").Should.Be.Enabled;
  Control.Named("bodyPanel").Should.Be.Hidden;
  ComboBox.Named("names").Should.Have.Items(new string[]{"Jeremy", "Scott", "Karl", "Jeffrey"});
  Select("Jeremy").For.ComboBox.Named("names");
}
```

13 Event Aggregator

Event Management is Hard

A direct quote from my tester today: "user interfaces are complicated critters." From time to time I bump into the idea that Graphical User Interface (GUI) programming is easy stuff suitable for keeping the amateurs busy while the adults are busy on the server side. My previous job involved very complex business logic behind a trivial GUI, but in systems like my current project the complexity of the GUI dwarfs the web services and database. There are several things that potentially make GUI programming hard, but the worst culprit by far to me is the event driven nature of heavy, interactive clients. A close second might be the synchronization of state between different parts of the user interface. Here's an example to illustrate what I mean:

Several years ago I found myself absurdly underused at work* with a brand new copy of the Gang of Four book on my bookshelf. I naturally decided to teach myself design patterns by writing a fullblown grid and reporting system for DHTML written in

JavaScript. The easy stuff was easy, but things suddenly got harder when I started to put these pieces together:

- A grid control. Given some data, an array of Column objects and ColorCoding objects, and a page size, make an HTML table on the fly. Allow sorting from the grid
- A pager control
- A filter control that allowed you to filter data in the grid based on the unique values of the data set. Think lot's of dropdowns
- A control that allowed you to customize the columns and color coding conditions
- A control that displayed the report header information

Fine and great, except I ran into two major problems. The first problem was that I needed to synchronize all of the various components of the logical "Report" anytime things changed (new data, filter changed, sorting, turning columns on and off, changing to a different report, etc.). What Martin Fowler calls flow synchronization turned out to be very clumsy. Besides the "n squared" communication problem, not every page had the exact same set of components. Having each component try to reach out and tell each other anytime the report state changed turned out to be a nightmare. Plus, what if I wanted to add completely new types of components to new report screens?

My second problem was the cascading event problem that's so painfully common in stateful GUI programming. Resetting the data in the grid causes the filter control to refresh it's dropdown boxes, which could easily fire off the onchanged event of the dropdowns, which would fire off a command to reset the grid data, which could... ...ripple all over the place.

I came to the conclusion that I needed to funnel all the events to a single class that would be responsible for propagating the events to all the other class's, and also to control the event propagation somewhat to shut down the rippling event problem. New components could be added to the screen and take part in the reporting workflow just by being registered as listeners to my new "ReportManager" class. What I came up with in the end was an implementation of the Event Aggregator design pattern.

The Event Aggregator pattern is essentially a Publish/Subscribe infrastructure inside your WinForms application. The Event Aggregator serves the same basic purpose as a messaging broker in a hub and spoke messaging architecture. If you have any experience (that isn't repressed memory) with publish/subscribe messaging architectures like webMethods or Tibco this should be old news. We can take much of the writings and patterns of loosely connected messaging and apply those lessons to the construction of composite user interfaces. To handle all these events flying around we can compose our system into three groups:

- Subscribers that need to be told about events
- Publishers that raise the events in the first place
- A "Hub" that collects all of the events from the publishers and routes and/or transforms the events to the proper subscribers

By channeling all of the events into a single hub we make it much easier to add new components to our user interface. This is absolutely crucial for the kinds of composite applications we're trying to build now with WinForms solutions. **Example Event Aggregator in StoryTeller**

Here's the setup. StoryTeller is, or will be shortly, a tool for editing and running acceptance tests with the FitnesseDotNet engine. The WinForms client is meant to replace the FitNesse Wiki for running tests, and as you can easily imagine, there are a lot of screen updates when test changes or is executed. Potentially, each test execution could effect:

- The TreeView control in the left pane that presents the Test/Test Suite hierarchy. The TreeView nodes that represent tests change their icon to reflect the test's status as unknown, successful, failed, or exception. Suspiciously similar to the NUnit GUI and every graphical test runner you've ever seen in your life.
- If it's open, the screen for editing a test. Some options are only enabled when there is a previous result. Visual cues need to change based on the result or state of the test. When the test is changed I gray out the title bar. When a test fails you get lots of red in the screen to indicate failure.
- Test grid pages. I wanted the grid to update (turn the grid row red or green, update the counts, etc.) when the Tests change state.

All three of these UI elements need to be updated on the same events, but it would be foolish to tie all of these pieces together with the nonvisual code that actually executes the tests. In Fowler speak, Flow Synchronization would lead to spaghetti code and undesirable coupling. Instead, I used a centralized Event Aggregator class to perform Observer Synchronization.

Most of the literature on using or building Event Aggregator's in WinForms applications revolves around using "Attribute magic" to define event subscribers and publishers. I understand a little now about why people were so upset or dubious with my "roll your own CAB" statements. The generic way CAB does Event Aggregator *is* complicated (but well within the reach of mere mortals if you were so inclined to do it). When I say that you don't need the CAB it's not because I think I can quickly roll my own CAB, it's because I think I can write little specific point solutions and quite possibly be better off for it.

My advice is to avoid the generic one size fits all Event Aggregator implementation for something specific to your own application with interfaces that are expressed in the terms of your problem domain. Make your Event Aggregator publish and subscribe events that are specific and meaningful to your problem domain. By making the events specific, the semantics of the code should be far more intention revealing than code like EventBroker.RaiseEvent("event://some string/more string/event name", new GenericEventArgs()) can ever be. I'm plenty happy to write a little bit of custom, specific code if the payoff is better understandability.

For StoryTeller, I took a very simple, direct path to creating an event aggregator for testing events. Let's ignore the publisher and hub pieces of event aggregation for now. The subscribers in StoryTeller need to expose and implement an Observer interface called ITestObserver with four methods:

```
    public interface ITestObserver
    {
        void StateIsUnknown(Guid testId);
        void StartedRunning(Guid testId);
        void Completed(TestResult result);
        void Queued(Guid testId);
    }
```

What I was aiming for with this design was semantic clarity and a strong contract. From an implementation standpoint it becomes pretty easy to organize the code that responds to test events. The first class to get the ITestObserver treatment was the TreeNode subclass for StoryTeller Test's:

public class TestNode : LeafNode<Test>, ITestObserver

TestNode is a descendent of TreeNode that represents a Test within the TreeView explorer view. TestNode changes it's display when any of the ITestObserver methods are called. For example, when a test run starts the StartedRunning() method is called. The TestNode will change its icon to a question mark to denote that the outcome is unknown, and change the TreeNode.Text to "name of the test (Executing)" to give some visual indication of which test is currently being executed. Here are a couple of the observer methods are just this:

```
        public void StateIsUnknown(Guid testId)
        {
            _lastResult = null;

            _context.Send(
                delegate
                    {
                        Text = Subject.Name;
                        RefreshImages();
                    }, null
                );
        }

        public void StartedRunning(Guid testId)
        {
            _lastResult = null;

            _context.Send(
                delegate
                    {
                        changeIcon(StoryTellerConstants.IGNORED);
                        Text = Subject.Name + " (Running)";
                    }, null
                );
        }
```

The TestNode just changes its text and icon to reflect the status of the Test. Since we've made it clear (I hope) that we don't want the test execution engine to "know" about the TreeView in the main screen, and the TestNode's certainly don't execute the Test's themselves, something else has to tell the ITestObserver's about Test events. The "hub" for publishing and subscribing to Test events in StoryTeller is an interface called ITestEventPublisher shown below:

```
[PluginFamily"("Default, Scope = InstanceScope.Singleton)]
public interface ITestEventPublisher
{
    void RegisterForTestEvents(ITestObserver observer, Test test);
    void UnRegister(ITestObserver observer, Test test);
    void PublishResult(Test test, TestResult result);
    void MarkAsQueued(Test test);
    void MarkAsUnknown(Test test);
    void MarkAsExecuting(Test Test);
    TestResult GetLastResult(Guid testId);
}
```

The [PluginFamily] attribute decorating the interface is just declarative wiring configuration for StructureMap (you don't have to use attributes for configuration). Part of the StructureMap configuration is to only create a single instance of ITestEventPublisher for the entire AppDomain. Anytime another class retrieves ITestEventPublisher through service location or ITestEventPublisher is injected into a constructor function the same single instance will be returned. In effect, we're just letting StructureMap handle the Singleton mechanics for us. It is, however, of the utmost importance to depend on the abstracted ITestEventPublisher interface rather than the concrete implementation (Chill out on the Singleton Fetish).

If you'll note the UnRegister(ITestObserver, Test) method for a second. In this particular case I have reasons to explicitly unregister listeners. In normal usage I'm having each screen unregister itself when it's closed to make sure that the garbage collection can reclaim the screen. Most people are taking advantage of WeakReference's to solve the garbage collection problem (if the publisher keeps a reference to each observer, the observers can not be garbage collected).

You'll also note that the RegisterForTestEvents(ITestObserver, Test) method explicitly requires you to specify the Test (the event subject) that the listener is interested in. In this case I'm making the concrete implementation of ITestEventPublisher responsible for remembering who's listening to which tests. Otherwise, you could have the publisher broadcast messages to each subscriber and just assume that the subscribers will be responsible for ignoring irrelevant events. I've never liked that approach in real messaging infrastructure, but I don't see it being that big of a difference inside a single process. In the StoryTeller case I thought that it made the implementation simpler to make the subscribers dumb.

Back to registering subscribers. Any class that listens for test execution events needs to register itself for each test with the an instance of ITestEventPublisher. The TestNode class registers itself in it's constructor function by retrieving the configured instance of ITestEventPublisher from StructureMap and passing itself into the RegisterForTestEvents(ITestObserver, Test) method:

```
        public TestNode(Test subject) : base(subject.Name, subject)
        {
            // Other setup that isn't relevant to the Event Aggregation…

            // Grab the Event Aggregator out of StructureMap and register
            // for any events related to the Subject of the node
            ITestEventPublisher publisher = ObjectFactory.GetInstance<ITestEventPublisher>();
            publisher.RegisterForTestEvents(this, subject);
            // Simple initialization
            StateIsUnknown(subject.TestId);
        }
```

If you've been following Scott Bellware's posts on dependencies, you'll recognize ITestEventPublisher as a transitive dependency of the TestNode. Once TestNode registers itself with ITestEventPublisher there's no more need to keep a reference around.

The internals of the concrete TestEventPublisher aren't particularly that complicated. The registered subscribers are stored in a Dictionary like this**:

```
        private Dictionary<Guid, List<ITestObserver>> _observers
            = new Dictionary<Guid, List<ITestObserver>>();
```

The methods for raising events to the subscribers look like these two below. Just find the ITestObserver's that are interested in a particular Test and fire the appropriate method for each subscriber.

```
        public void MarkAsQueued(Test test)
        {
            test.Status = TestStatus.Queued;
            foreach (ITestObserver observer in GetObservers(test))
            {
                observer.Queued(test.TestId);
            }
        }

        public void MarkAsUnknown(Test test)
        {
            test.Status = TestStatus.Unknown;
            foreach (ITestObserver observer in GetObservers(test))
            {
                observer.StateIsUnknown(test.TestId);
            }
        }
```

Inside of the unit tests for TestEventPublisher I use RhinoMocks to test the interaction of the TestEventPublisher with the ITestObserver's. The first set of unit tests just checks that TestEventPublisher is correctly associating ITestObserver's with the proper Test subjects (I'll spare you the details). After that I wrote tests like the following that verifies that the ITestObserver.Completed(TestResult) method is fired on the correct ITestObserver's whenever ITestEventPublisher.PublishResult(Test, TestResult) is called.

```
        private TestEventPublisher setupMockedPublisher()
        {
            // Using a Self-Mock for TestEventPublisher
            // I've already unit tested the GetObservers(Test) method,
            // so I'd like to remove that little wrinkle from the other tests
            TestEventPublisher publisher = _mocks.CreateMock<TestEventPublisher>();

            Expect.Call(publisher.GetObservers(test1))
                .Return(_observerArray)
                .Constraints(new Equal(test1));

            return publisher;
        }

        [Test]
        public void PublishResult()
        {
            TestEventPublisher publisher = setupMockedPublisher();
            TestResult result = new TestResult();

            foreach (ITestObserver observer in _observerArray)
            {
                observer.Completed(result);
            }
            _mocks.ReplayAll();
            publisher.PublishResult(test1, result);
            _mocks.VerifyAll();
        }
```

Great, we've got an interface for subscribers and a unit tested hub (TestEventPublisher). Now all we need is to actually fire off the events from the publishers. Instead of opting for more indirection like the CAB, I just have the publishers call methods directly on the ITestEventPublisher interface. Again, I think it's simpler and makes the code more intention revealing (the CTRL-B factor as well). If you want more decoupling you could have the hub listen to events on the publishers and then relay and/or transform the events to the subscribers.

One way or another, the publishers get the single instance of ITestEventPublisher from StructureMap. The presenter for the screen that let's you edit a Test gets a reference from constructor injection.

```
        public TestEditorPresenter(ITestView view, ITestFormatConverter converter,
    ITestEventPublisher publisher,
        ICommandExecutor executor) : base(view, converter)
        {
            _publisher = publisher;
            _executor = executor;
        }
```

The publisher I want to look at is called ExecutionEngine. Internally, it's responsible for managing and coordinating the workflow of running tests. In this case ITestEventPublisher is a transparent dependency that ExecutionEngine interacts with throughout it's lifetime.

```
        public ExecutionEngine(ITestRunner runner, IComponentAnalyzer[] analyzers, ITestEventPublisher
            publisher)
        {
            _runner = runner;
            _analyzers = analyzers;
            _publisher = publisher;
        }
```

Deep inside of ExecutionEngine it has a private method that actually executes a batch of tests.

```
        private List<TestResult> executeTests(Test[] tests)
        {
            List<TestResult> list = new List<TestResult>();

            foreach (Test test in tests)
            {
                Console.WriteLine"(Running Test " + test.GetFullPath());
                _publisher.MarkAsExecuting(test);
                TestResult result = _runner.ExecuteTest(test);
                list.Add(result);
                _publisher.PublishResult(test, result);
                Console.WriteLine"( "  + result.Counts.ToString());
            }
            return list;
        }
```

ExecutionEngine calls the ITestEventPublisher.MarkAsExecuting(Test) method just before executing a Test and publishes the result afterwards with the ITestEventPublisher.PublishResult(Test, TestResult) method to let the subscribers know that an individual test is finished. This is important functionality. The StoryTeller tests can often be slow to execute because they're usually integration tests, and it's very handy to both give some UI cues about the progress and also allow the user to begin reviewing the test results while the remaining tests execute. It's a huge advantage over the FitNesse Wiki tool that I'm trying to replace with StoryTeller.

Part of the unit test for running a batch of tests is shown below. I'm using RhinoMocks "ordered" mode to validate that the interactions happen in the correct order.

```
            using (mocks.Ordered())
            {
                // Other stuff
                publisher.MarkAsExecuting(test1);
                Expect.Call(runner.ExecuteTest(test1)).Return(result1).Constraints(new Equal(test1));
                publisher.PublishResult(test1, result1);
                publisher.MarkAsExecuting(test2);
                Expect.Call(runner.ExecuteTest(test2)).Return(result2).Constraints(new Equal(test2));
                publisher.PublishResult(test2, result2);
                publisher.MarkAsExecuting(test3);
                Expect.Call(runner.ExecuteTest(test3)).Return(result3).Constraints(new Equal(test3));
                publisher.PublishResult(test3, result3);
                publisher.MarkAsExecuting(test4);
                Expect.Call(runner.ExecuteTest(test4)).Return(result4).Constraints(new Equal(test4));
                publisher.PublishResult(test4, result4);
                // Other Stuff
            }
```

So far I'd say that the TestEventPublisher design has worked out rather well. The one issue I didn't mention was thread synchronization as the TestEventPublisher could easily be running in a background thread instead of the UI thread. I'll leave that solution up to you dear reader;) On a serious note, do be aware of the thread synchronization issue here.

The rest of the code for the Event Aggregation in StoryTeller can be found in the Subversion trunk here: http://storyteller.tigris.org/svn/storyteller/trunk/src.

13.0.1 Event Aggregator with Generics

What's the Problem?

Ideally a single screen is basically self-contained. The View and Presenter talk to each other, the Presenter talks to some sort of service layer, and that's that. But like any idyllic spot, the real world will intrude. A screen will often need to react to actions or events taking place in other parts of the UI. You may have a feature that broadcasts updated data from the server to the client that would need to update the display of a screen or the state of the UI. Some action taking place in one screen might affect other parts of the UI, and it can easily be more than one element of the UI.

At this point we're facing a situation where a lot of screen elements, Views, and pieces of the application shell are interrelated and need to be synchronized. The problem we have right off the bat is how to get all of these pieces talking together in the first place. The next problem is how to avoid making the UI code a tangle of references from all these different parts trying to get a reference to each other just to talk to one another. If you're not careful, you can create some nasty spaghetti code connecting all the disparate parts of the UI.

Once again, we're going to reach for a Publish/Subscribe method for routing events from the origin of the event. To eliminate undesirable coupling between the event publishers and the event listeners we're going to route all the notifications through a central messaging hub called an Event Aggregator. If a publisher can simply send a message to the Event Aggregator hub, the hub will relay that message to the proper listeners. New listeners can be added later to the Event Aggregator without the publishers being affected.

The Scenario

Here's my scenario from StoryTeller. I have a pane on the left that acts in the same capacity as the solution explorer from Visual Studio to show all of the open testing projects and provide access to all of the Tests and Test Suites. When a user adds a new testing project to the working set, another part of the UI called the FixtureExplorer needs to update its display to reflect the available Fixture classes for each project. This isn't coded yet, but when a testing project is removed, I need all open tabs that are related to that project to be closed and removed from the UI.

The three main challenges I faced were

- How does the Event Aggregator know which listeners are interested in a given message?
- How to get a reference to the Event Aggregator?
- How to register the listeners?

Message Routing with Generics

I didn't want my listeners to have to implement a generic handler method that would have to analyze a message coming in to decide what to do. For the sake of easier testing and more explicit code, I wanted a strongly typed method as part of a listeners public API that handled a specific message. Furthermore, to simplify the listeners, I want the Event Aggregator to be responsible for deciding what messages a given listener is interested in. And oh by the way, I have to be able to add new message types without changing the Event Aggregator.

In the end, the answer was really pretty simple. The listeners just need to implement the IListener<T> interface, where T is the type of a message. The Handle() method is where a listener reacts to an event message.

```
    public interface IListener<T>
    {
        void Handle(T subject);
    }
```

For a concrete example of a listener, I have a class called FixtureExplorerPresenter that needs to listen to several events related to testing projects. To register as a listener, it just has to implement IListener for each event:

```
    public class FixtureExplorerPresenter : IFixtureExplorerPresenter,
            IListener<SystemLoaded>,
            IListener<SystemActivated>,
            IListener<SystemDeactivated>,
            IListener<SystemReloaded>,
            IListener<SystemRemoved>
```

The listening methods look something like this:

```
        // This is the implementation of the IListener<SystemLoaded> interface
        // The Event Aggregator will call this method when a SystemLoaded message
        // is published
        public void Handle(SystemLoaded subject)
        {
            ISystemUnderTest system = subject.SystemUnderTest;
            ReloadSystem(system);

            if (isSystemActive(system))
            {
                loadNode(system);
            }
        }
```

The details of what it's doing isn't that interesting, but I want to point out how easy it was in a listener to subscribe to an event. There's no generic method that receives an event and then has to run it through a switch statement to figure out what to do. FixtureExplorerPresenter just gets a SystemLoaded event and works accordingly. In my unit tests for FixtureExplorerPresenter I can just walk right up to the class and call Handle(SystemLoaded) without having to set up anything of the rest of the system.

```
        [Test]
        public void SystemLoadedReloadsTheFixturesForThatSystemAndResetsTheViewIfThisIsTheActiveSystem()
        {
            FixtureExplorerPresenter presenter = _mocks.CreateMock<FixtureExplorerPresenter>(_view, _writer);
            ISystemUnderTest system = ObjectMother.SystemUnderTest();
            presenter.ReloadSystem(system);
            Expect.Call(_view.ActiveSystemName).Return(system.Name);
            SystemFixturesNode theNode = new SystemFixturesNode(system, new FixtureToken[0]);
            Expect.Call(presenter.GetNode(system)).Return(theNode);
            _view.DisplayFixturesNode(theNode);
            _mocks.ReplayAll();
            presenter.Handle(new SystemLoaded(system));
            _mocks.VerifyAll();
        }
```

Creating the listeners was easy enough because all a listener had to do was implement IListener<T> interfaces for all the events it needed to listen to. We've still got an outstanding responsibility for publishing events to the correct listeners, so let's turn our attention to the hub in our publish/subscribe eventing model.

In StoryTeller, events are published to the rest of the system by getting a reference to an instance of this interface (highly elided) shown below;

```
    public interface ITestEventPublisher
    {
        ...
        void PublishEvent<T>(T subject);
        void AddListener(object listener);
        void RemoveListener(object listener);
    }
end{lstlisting}

A client of the ITestEventPublisher interface just needs to call the PublishEvent<T>(T message) method to broadcast a message to the rest of the application.  In one of the samples above I showed some code in the FixtureExplorerPresenter class that responded to a ""SystemLoaded event to update part of the screen.  Here's the other side of the publish/subscribe.  Another part of the system called ApplicationController is responsible for loading new testing Projects into the test explorer.  The ApplicationController.AddProject() method shown below needs to tell FixtureExplorerPresenter and the other listeners that a new ""SystemUnderTest has been loaded into the UI.  Instead of calling FixtureExplorerPresenter directly and causing the n^2 messaging anti-pattern, it sends a SystemLoaded message to an ITestEventPublisher.
\begin{lstlisting}
        public virtual void AddProject(IProject project)
        {
            ISystemUnderTest system = project.BuildSystem();
            _hierarchyNode.AddProject(project, system);
            // _publisher is an instance of the ITestEventPublisher interface
            _publisher.PublishEvent(new SystemLoaded(system));
        }
```

The concrete method PublishEvent() is shown below:

```
        public void PublishEvent<T>(T subject)
        {
            // _listeners is just an ArrayList of all the Listeners
            foreach (object listener in _listeners)
            {
                // Determine if a Listener handles the message of type T
                // by trying to cast it
                IListener<T> receiver = listener as IListener<T>;
                if (receiver != null)
                {
                    // I'm using SyncronizationContext to handle moving processing
                    // from a background thread to the UI thread without having
                    // to worry about it in the View or Presenter
                    _context.Send(delegate { receiver.Handle(subject); }, null);
                }
            }
        }
```

Connecting a message to the correct listeners is actually pretty simple. The TestEventPublisher object as an ArrayList of all the possible listeners. When it receives a message of type SystemLoaded, TestEventPublisher:

- Loops through each listener
- Checks if each listener implements the interface IListener<SystemLoaded>
- If it does implement that interface, it calls Handle(SystemLoaded) on the listener

That's it. Assuming that the TestEventPublisher "knows" about all the listenters, and a client of ITestEventPublisher "knows" how to get to the single instance of TestEventPublisher, event aggregation is easy. So let's talk about getting the pieces together.

Getting Listeners and Publishers Together

The easy part is just getting a reference to the single instance of the ITestEventPublisher. That's just a little bit of vanilla StructureMap configuration:

```
            ForRequestedType<ITestEventPublisher>()
                .TheDefaultIsConcreteType<TestEventPublisher>().AsSingletons();
```

The bolded part just directs StructureMap to only create one shared instance of ITestEventPublisher. Anything that needs to get at an ITestEventPublisher can just service location by a good ol'fashioned call to:

```
            ITestEventPublisher eventPublisher =
                ObjectFactory.GetInstance<ITestEventPublisher>();
```

if you only need a transient reference to ITestEventPublisher. The mass majority of the time I just use constructor injection like my ApplicationController class:

```
        public ApplicationController(
            ICommandExecutor executor,
            IContentPanel panel,
            IProjectRepository repository,
            ITestEventPublisher publisher,
            IHierarchyNode hierarchyNode)
        {
            _executor = executor;
            _panel = panel;
            _repository = repository;
            _publisher = publisher;
            _hierarchyNode = hierarchyNode;
        }
```

I do create ApplicationController from StructureMap, so all of the constructor arguments are located and passed into the constructor by StructureMap. All of the Command and Presenter classes that would need to broadcast event messages are generally created by StructureMap anyway, so the constructor injection method turns out to be pretty simple.

All that's left is to make sure that the listeners are correctly registered with the TestEventPublisher, and that's a little bit trickier. Before I start, I should say that this is just the way that I did this and not THE way to do this. In StoryTeller there are two different categories of listeners. There is one set of objects like the FixtureExplorerPresenter that are driving pieces of the application shell at all times. These objects are started as the application loads and registered with TestEventPublisher right off the bat. I'm using the fluent interface mechanism for configuring StructureMap in code as a more or less internal DSL (I refuse to argue with you if it's actually a DSL or not). One of the advantages of an internal DSL written in just plain C# is that you can extend it at will or wrap it with other calls. I created a little wrapper around the method to register a type with StructureMap called Start():

```
        private static List<Type> _startables = new List<Type>();
        protected CreatePluginFamilyExpression<T> Start<T>() where T : IStartable
        {
            Type type = typeof (T);

            // Just track which Types need to be started and registered with
            // ITestEventPublisher as the application is loaded
            _startables.Add(type);
            // Continue on with StructureMap configuration as usual
            return ForRequestedType<T>();
        }
```

In the StructureMap bootstrapping, I just call Start() to denote a type that needs to be started right off the bat and registered with the event aggregator. Notice that our old friend FixtureExporerPresenter is one of the "Startables."

```
            Start<IFixtureExplorerPresenter>()
                .TheDefaultIsConcreteType<FixtureExplorerPresenter>().AsSingletons();

            Start<IApplicationController>()
                .TheDefaultIsConcreteType<ApplicationController>().AsSingletons();

            Start<ICommandExecutor>().TheDefaultIsConcreteType<CommandExecutor>().AsSingletons();
```

Part of the application bootstrapping is a call to this little routine. It just iterates through the “Startable” types, gets that type from StructureMap, and registers each by calling AddListener() on the event aggregator.

```
        private static void attachListeners()
        {
            ITestEventPublisher eventPublisher =
                ObjectFactory.GetInstance<ITestEventPublisher>();

            foreach (Type type in _startables)
            {
                IStartable startable = (IStartable) ObjectFactory.GetInstance(type);
                startable.Start();
                eventPublisher.AddListener(startable);
            }
        }
```

I also register the Presenter for each screen as a listener each time a new screen is opened. I’ll talk much more about this in a future post, but for right now let’s say that all screens are opened by a single class (sometimes it’s the ApplicationController, other times it’s what I call a ScreenCollection for a lack of a better name). By making all screen activation and deactivation go through a common point it’s now easy to perform bookkeeping operations on the screens. In the case of StoryTeller, I do this in the method that’s called to add a new screen to the main docking manager (the code is in ContentPanel if you’re playing along at home in the StoryTeller code):

```
        public void OpenScreen(IPresenter presenter)
        {
            _publisher.AddListener(presenter);
            //Other stuff that adds the screen to the docking manager
       ...
        }
```

I mentioned above that I don’t use WeakReferences. To avoid a severe memory leak in the app, I need to remove a presenter from the publisher’s list of listeners whenever a screen is permanently closed.

```
        public void Close(ContentTab tab)
        {
            bool needToActivateDifferentScreen = tab.Equals(TabControl.SelectedTab);
            IPresenter presenter = tab.Presenter;
            presenter.Deactivate();
            _publisher.RemoveListener(presenter);
            _tabControl.Items.Remove(tab);

            if (needToActivateDifferentScreen && _tabControl.Items.Count > 0)
            {
                ContentTab lastTab = getLastTab();
                TabControl.SelectedTab = lastTab;
            }
        }
```

If you’re not going to track the elements that need to be removed from the publish/-subscribe registry, or just don’t want to do this, I’d recommend using WeakReferences inside your Event Aggregator class. **Other Issues**

Some last little thoughts.

- I’m using SynchronizationContext for threading synchronization so that I don’t have to worry about events coming from a background thread. I’ve found that choking the background callbacks through just a couple different points can keep the rest of your code cleaner from the monotonous InvokeRequired()/Invoke() code.

- I don't think it matters in StoryTeller, but with other applications you may not want to immediately respond to an event if the screen is hidden or in an inactive tab. You'll need more logic in these cases to delay an update activity.

There's probably other issues, but I want to hit "publish" now. Just throw in questions and I'll see what I can do.

14 Rein in runaway events with the "Latch"

The "Latch"

In the last post I talked a little bit about runaway events in a fat client. One event causes an action that changes another control which conceivably fires another event which eventually sends you application spiraling out of control. All because a little bug went Kachhhoooooo! What we need to do is ignore or turn off an event handler while a certain action is taking place. You could just temporarily detach the event handler while you're performing that action like this code:

```
            // Remove the event handler for the moment
            _someComboBox.SelectedIndexChanged -= new System.EventHandler(someHandler);
            // do something that would probably make _someComboBox fire the SelectedIndexChanged event
            // Put the event handler back
            _someComboBox.SelectedIndexChanged += new System.EventHandler(someHandler);
```

It works, but I wouldn't want to do it everywhere. It's tightly coupling the code inside this method with the rest of the View. I would bet that this approach would quickly make a View very difficult to modify.

My preferred approach is what I call a "Latch" (it's taken from a pattern used in messaging to stop a message from endlessly cycling between two systems that publish events to each other). Inside any method or logical operation that could cause cascading events you simply set the latch to mark an operation in progress. In the relevant event handlers you first check to see if the latch is set, and cancel any additional actions in the event handler is the latch is indeed set. It could be as simple as just tracking a Boolean field in a View class. Slightly more sophisticated is something like the "Latch" class from StoryTeller:

```
public delegate void VoidHandler();
public class Latch
{
    private int _count = 0;

    public void Increment()
    {
        _count++;
    }

    public void Decrement()
    {-
        _count;
    }

    public bool IsLatched
    {
        get { return _count > 0; }
    }

    public void RunInsideLatch(VoidHandler handler)
    {
        Increment();
        handler();
        Decrement();
    }

    public void RunLatchedOperation(VoidHandler handler)
    {
        if (IsLatched)
        {
            return;
        }
        handler();
    }
```

There isn't too much to the usage. In the method that performs work you might do this:

```
      _latch.RunInsideLatch(delegate
{
  // The actions that spawn cascading events
  activatePresenter(presenter, page);
  _tabControl.Items.Add(page);
  _tabControl.SelectedTab = page;
} );
```

In an event handler effected by this work you could guard the event propagation by doing this

```
void TabControl_TabSelected(object sender, TabEventArgs args)
{
    if (_latch.IsLatched)
    {
        return;
    }

    ContentTab tab = (ContentTab) TabControl.SelectedTab;
    activatePresenter(tab.Presenter, tab);
}
```

or this version (fun with anonymous delegates):

```
        void TabControl_TabSelected(object sender, TabEventArgs args)
        {
            _latch.RunLatchedOperation(
                delegate
                    {
                        ContentTab tab = (ContentTab)TabControl.SelectedTab;
                        activatePresenter(tab.Presenter, tab);
                    });
        }
```

I've used this pattern about 3-4 times with some success. It's especially powerful in conjunction with an Event Aggregator.

So what did I really buy here? In the application, opening a new screen involves the creation and activation of a new TabControl, which raises an event for selecting a new tab. For tabs that are just inactive, I need an indication from the SelectedTab event of a TabControl to tell that particular Presenter to reactivate. The SelectedTab event is necessary for screens that are already open, but nothing but trouble for creating a brand new Tab. By setting a latch in the event that opens a new TabControl, I an easily ignore the SelectedTabl event just while the TabControl is being setup. **Is this really a design pattern?**

I've used some derivation of this solution on at least four projects, so it's safe to say that it is *a* pattern. Design patterns generally aren't anything terribly fancy, there just things that you do – repeatedly. When experienced people learn design patterns I often hear some snorting to the effect of "I already do this." Yes, that's kind of the point in calling it a "pattern."

I'm not an officially sanctioned pattern naming body, so it's not necessarily appropriate for me to be making up the name here. I'd bet anything that someone else has already described this pattern and I'm just not familiar with the named pattern. If you've seen this written up somewhere else, please throw in a link in the comments.

15 Embedded Controllers with a Dash of DSL

Why is this post necessary?

Why, you might ask, are you writing a post on what amounts to a "ViewHelper?" One simple reason – View's can easily become absolutely massive blobs of code. Any chance to move a cohesive set of screen responsibilities into another class should serve to make the View itself simpler, and that's all this post is about. Plus, breaking a View's behavior into multiple, cohesive classes can potentially lead to reuse opportunities for the little Embedded Controller classes.

For much of the last three years I've worked with a lot of legacy code over a half-dozen different codebase's. All of them, to be charitable, were less than ideal in quality and structure. If you ask me what the single biggest flaw or problem across all of these codebase's my answer would be near automatic — long classes and long methods. What

I continuously see is code stuffed into modules until the modules are coming apart at the seams.

Come to think of it, my current project is about 95**Embedded Controller**

Again, I'm not a sanctioned patterns naming body, but the term "Embedded Controller" is my name for nonvisual classes that help a View control some distinct part of it's behavior. The first example that comes to my mind is from a WinForms project that used a 3rd party grid (not a vendor on the CodeBetter friends list by the way). The grid control needed a lot of consistent help and infrastructure code around it to implement the behavior we needed (little things like sorting and paging). We quickly discovered that additional screens needed the exact same bootstrapping code, so the obvious answer was to extract that "grid helper" into it's own reusable class. We were using a pretty strict Passive View approach, but even so, we didn't want the Presenter's tied that tightly to the screen mechanics. Instead, the new GridHelper class was just something that the View controls used internally. After the third screen with the grid control, development suddenly went faster.

To differentiate Embedded Controller classes somewhat from the Presenter, here's the attributes of an Embedded Controller:

- Nonvisual class used by a View to implement some of the View behavior. I guess in
- Completely encapsulated within a View. There is no sign of the Embedded Controller in a View's publicly facing interface.
- Embedded Controller's are happily aware of the actual, concrete UI widgets. The Embedded Controller class "knows" about buttons and checkbox's and the nasty 3rd party grid that you're being forced to use.
- Very limited in scope. An Embedded Controller provides classical controller functionality for a very specific part of the screen

My advice for taking advantage of Embedded Controller's is threefold:

- Look for common UI coding tasks within a system and look for opportunities to encapsulate some screen mechanics in reusable pieces. This is just another exercise in eliminating duplication.
- Split up any View class that gets too big. I might have left the impression in earlier posts that the View code is somehow exempt from the normal coding standards because we've made it "simpler" now. Code quality matters everywhere, and especially in areas of the code that are likely to change over time — like View's. Even with a Passive View architecture a complex screen will almost inevitably lead to complex code in the View.
- Pulling out an Embedded Controller might be an easy way to extend unit testing deeper into the View. This won't always be true, but a "POCO" embedded controller class can often be quite easy to unit test inside vanilla xUnit tests without

resorting to anything exotic like NUnitForms. There is some widget behavior that only functions when a Form is visibly displayed, but a lot of behavior can be tested just by instantiating UI widgets directly within a unit test. One way or another the UI widget stuff is nothing but CLR classes.

Sample: The Control State Machine

Here's a scenario from my current project that I bet all of us have dealt with a few times over. We have a Trade screen that has seven different states depending upon whether you're creating or reviewing a Trade. The various user actions available on the screen differ from state to state. As the screen changes state either upon opening the screen or a result of user actions while it's open we need to enable/disable and show/hide different screen elements. The screen started simple, so I just coded specific methods at first to enable or disable bits of the screen. Fast forward a couple of weeks and the behavioral logic had exploded (funny how that happens when you actually get to talk to the end users). Unsurprisingly, the code in the Presenter and View had become hairy, plus the screen had way too much flicker for that matter.

Before I show any code, let's be pretty clear that this code is not very optimized or even very powerful. All I want to talk about is the concepts and structure of the design irrespective of performance.

At least in concept, the solution was pretty simple. Move that logic into a state machine construct. Since we already had a full set of regression tests against the UI screen itself, I felt pretty safe making the changes. All I did was create a class called "ControlState" that's nothing but a collection of Control's to display and enable for a particular screen state. ControlState has a method called Activate() which simply loops through its internal collection to enable and show the configured controls (it's not shown but the call to Activate() is wrapped in SuspendLayout() and ResumeLayout()).

```
         public class ControlState
        {
            private List<Control> _displayedItems = new List<Control>();
            private List<Control> _enabledItems = new List<Control>();

            public void ShowControls(params Control[] controls)
            {
                _displayedItems.AddRange(controls);
            }

            public void EnableControls(params Control[] controls)
            {
                _enabledItems.AddRange(controls);
            }

            public void Activate(ControlStateMachine<T> machine)
            {
                machine.LevelSet();
                foreach (Control item in _enabledItems)
                {
                    item.Enabled = true;
                }
                foreach (Control item in _displayedItems)
                {
                    item.Visible = true;
                }
            }
        }
```

As you can probably guess, there's a second class that aggregates all of the configured ControlState's and Control's called ControlStateMachine<T>, where T is an enumeration of the possible states. Here's a little bit of its implementation.

```
        private readonly Control _parent;
        private readonly IScreenBinder _binder;
        private List<Control> _displayedItems = new List<Control>();
        private List<Control> _enabledItems = new List<Control>();
        private Dictionary<T, ControlState> _states = new Dictionary<T, ControlState>();
        private T _currentState;

        public ControlStateMachine(Control parent, IScreenBinder binder)
        {
            _parent = parent;
            _binder = binder;
        }

        public void SetState(T stateKey)
        {
            _parent.SuspendLayout();
            _states[stateKey].Activate(_binder, this);
            _currentState = stateKey;
            _parent.ResumeLayout();
        }
```

The View itself just calls ControlStateMachine.SetState() to configure itself.

Now, for the cool part. Here's a somewhat obfuscated version of our code that defines the state machine inside the View.*

```
        private void configureStateMachine()
        {
            _stateMachine = new ControlStateMachine<TradeDetailState>(this, _binder);
            _stateMachine.OnStateChangeTo(TradeDetailState.Creation)
                .Show(createTradeButtonsPanel)

        .Enable(status1CheckBox,
    status2CheckBox, status3CheckBox, externalTradeIdTextbox);

            _stateMachine.OnStateChangeTo(TradeDetailState.Review)
                .Show(updateTradeButtonsPanel)
                .IsReadOnly()
                .Enable(
                    editTradeButton,
                    status1CheckBox,
                    status2CheckBox,
                    status3CheckBox,
                    externalTradeIdTextbox,
                    cancelTradeButton);

            _stateMachine.OnStateChangeTo(TradeDetailState.ReviewDirty)
                .Show(updateTradeButtonsPanel)
                .IsReadOnly()
                .Enable(
                    undoButton,
                    status1CheckBox,
                    status2CheckBox,
                    status3CheckBox,
                    submitTradeChangesButton,
                    externalTradeIdTextbox,
                    cancelTradeButton);

            _stateMachine.OnStateChangeTo(TradeDetailState.Edit)
                .Show(updateTradeButtonsPanel)
                .Enable(externalTradeIdTextbox, cancelTradeButton);

            _stateMachine.OnStateChangeTo(TradeDetailState.EditDirty)
                .Show(updateTradeButtonsPanel)

        .Enable(undoButton,
submitTradeChangesButton, externalTradeIdTextbox, cancelTradeButton);

            _stateMachine.OnStateChangeTo(TradeDetailState.Cancelled)
                .Show(updateTradeButtonsPanel)
                .DisableEverything()
                .IsReadOnly();

            _stateMachine.OnStateChangeTo(TradeDetailState.History)
                .Show()
                .DisableEverything()
                .IsReadOnly();
        }
```

If you haven't seen this kind of syntax before, it's what Martin Fowler (a real patterns naming authority) has christened Fluent Interface. Why, oh why, did I go to the extra trouble of making a Fluent Interface instead of just defining the state machine by filling up the collection state (and let's be clear, it is a little more work)? Because I thought it would make a good blog post wanted to create Domain Specific Language in the code to make the code easier to understand.

That's Not Really a DSL!

Unfortunately, in my opinion, the .Net community is fixated on graphical Domain Specific Language's (DSL) that revolve around yet more code generating visual tooling.

There's a complete other side to the DSL's however. Another alternative is lexical languages which could be either internal/embedded or external to the language, with the pro-lexical argument being something like "people can read English you know." I'm not particularly enamoured of creating my own programming language and interpreter, so my particular area of interest right now is in creating embedded DSL-like syntax's inside existing languages. Granted, C# is very limited in this respect compared to other languages, but we can still achieve some useful gains with Fluent Interface coding.

We're about to get IronRuby and C# 3 as fullblown CLR languages. Both languages, and especially IronRuby, are far better suited for internal DSL's than C# 2.0. The Ruby hype seems to be 9/10's Rails, but the stuff that ThoughtWorks and others are doing with expressive DSL development in Ruby might turn out to be the real value proposition behind Ruby.

So is my little ControlStateMachine a DSL or just an API? Running it through the Is It a DSL or an API Checklist, the answer is probably no, but the real value is simply an expressive AP. The real goal (or enabler) is to move toward coding syntax's and API's that speak clearly in the semantics of the problem domain. The general idea is that the correctness, or intent, of the code should become much more readily verified through simple inspection. Ideally, you could move to the point where you are able to show your business logic code to the actual business users. There will still be plenty going on behind the scenes, but they don't need to see that.

David A. Black has a good post on Domain Specific Language that I'd recommend for a jumping off point.

I haven't delved deeply enough yet, but Boo sounds like a good way to create lexical DSL's in .Net today.

16 Managing Menu State with MicroController's, Command's, a Layer SuperType, some StructureMap Pixie Dust, and a Dollop of Fluent Interface

Problem Statement

Again, I am not a licensed namer of patterns, and some of the people I've shown this code to have commented that it's reminiscent of Smalltalk UI's, so it's a good bet that some of you have already seen or used similar approaches. That said, I use the term "MicroController" to refer to a controller class that directs the behavior of a single UI widget. By itself, a MicroController isn't really that powerful, but like an army of ants, a group of MicroController's working cooperatively (with some external direction) can accomplish powerful feats.

Here's a common scenario:

- You have a menu bar of some sort or another in the top strip of your composite WinForms application

- Each individual menu item needs to fire off a command of some sort (duh)
- The availability of some, but not all, menu items is dependent on user roles. In other words, some menu items will be disabled or hidden if a user does not have a necessary role.
- The availability of some menu items is dependent upon the screen that is currently active in an application with some sort of MDI or tabbed interface
- The menu bar frequently collects new additions as the application grows

There's nothing in that list that's particularly hard or even unusual, but I want to explore an alternative approach. Off the top of my head, I've got four goals in mind for the design of my menu state management:

- Quickly attach the proper commands and actions to each MenuItem. This can get tricky because each MenuItem potentially touches and interacts with very different pieces of the application.
- Eliminate duplicate grunge coding. I say you pursue any chance to eliminate the mechanical cost, and there's quite a bit of repetitive functionality here.
- Make the code that specifies the menu behavior as expressive and readable as possible. This code is going to frequently change, so it's worth our while to make this code readable.
- I want an easy way for each screen to configure the menu state, and I want this menu state calculation and manipulation to be as easy to understand, write, **and test** as possible.

Much like the screen state machine sample from my last post, I'm going to try to use a Domain Specific Language (DSL) / Fluent Interface to express and define the menu behavior. By hiding the mechanics of menu management behind an abstracted Fluent Interface I'm hoping to compress the code that governs the menu state to a smaller area of the code. I want to be able to understand the menu behavior by scanning a cohesive area of the code. It's my firm contention that this type of readability simply cannot be accomplished by using the designer to attach bits and pieces of behavior. Leaning on the designer will scatter the behavior of the screen all over the place. One of the main reasons I don't like to use the designer or wizards is because you often can't "see" the code and the way it works.

Start with the End in Mind

Before zooming in one the individual components of the solution, let's keep the man firmly behind the curtain and look at my intended end state. Inside some screen (probably the main Form) is a piece of code that expresses the behavior of the menu items like this fragment below:

```
        private void configureMenus()
        {
            _menuController.MenuItem(openItem).Executes(CommandNames.Open).IsAlwaysEnabled();
            _menuController.MenuItem(saveItem).Executes(CommandNames.Save)
                .IsAvailableToRoles("BigBoss", "WorkerBee");

            _menuController.MenuItem(executeItem).Executes(CommandNames.Execute)
                .IsAvailableToRoles("BigBoss");

            _menuController.MenuItem(exportItem).Executes(CommandNames.Export);
        }
```

And in each individual screen presenter you might see some additional code to set screen-specific menu settings like this that would be called upon activating a different screen:

```
    public class SomeScreenPresenter : IPresenter
    {
        public void ConfigureMenu(MenuState state)
        {
            state.Enable(CommandNames.Save, CommandNames.Export);
            state.Enable(CommandNames.Execute).ForRoles("SpecialWorkerBee", "BigBoss");
        }
    }
```

So what's going on here? There isn't a single call in this code to MenuItem.Enabled or any definition of MenuItem.Click, so it's safe to assume that there's somebody behind the curtain. So, what is the man behind the curtain? Before I talk about each piece in detail, here's a rundown of the various moving parts:

- A small MicroController object for each MenuItem that "knows" when to enable or disable its MenuItem and set up the MenuItem's Click event.
- A controller class for the menu that aggregates all of the MenuItem controllers. Part of the responsibility of the Fluent Interface configuration code above is to associate a MenuItem with a CommandNames key so we can quickly reference the correct MenuItem's when a different screen is activated.
- A series of Command pattern classes that all implement a common interface.
- StructureMap is lurking in the background as my IoC tool of choice to wire the various parts together.
- The Fluent Interface configuration API. For the most part, it's job is to look pretty. All it's doing is gathering in input values to set on the individual MenuItem controller classes that do the real work. What I've found so far is that Fluent Interface strategies seem to lend themselves best to situations like this where you're really just configuring an object graph to do the real work.
- A MenuState object that is used to transmit the desired state of the menu from the individual screens to the main menu controller
- Each Presenter in the application implements a common interface that includes some sort of hook to configure the items on the main Menu.

- The whole application is stitched together with the combination of an ApplicationController and ApplicationShell. I'm going to be very brief on these two because I'm going to devote a couple posts later down the line to these classes.

17 MicroControllers

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.

- 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
- 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. 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.

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.

```
        // 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(strategyLabel)
                .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(legalEntityLabel)
                ;
            _binder.BindProperty(TradeFields.CounterParty).To(counterPartyCombobox).WithLabel(
                counterPartyLabel)
                .FillWith(ListType.Counterparty);

            _binder.BindProperty(TradeFields.TradeDate).To(tradeDatePicker).WithLabel(tradeDateLabel)
                .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<ExerciseTypeEnum>()
                .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<FakeEnum>());
    }

    [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(strategyLabel)
                .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:

Loop through each IScreenElement Query the Notification object for all of the error messages for the IScreenElement.FieldName 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, checkBox)
        {
            checkBox.CheckedChanged += checkBox_CheckedChanged;
        }

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

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

    }
```

18 Boil down the "wiring" to a Registry

A large part of building a composite application is "wiring" the pieces together. You're wiring views to presenters, commands to menus, and security rules to elements of the screen. Since you're trying to make this work faster, you make a lot of this work declarative in nature. Great. Now let's talk about how you do that wiring. You can use some attributes, xml configuration, or use some sort of in code registry. All of that declarative wiring configuration is the "story" of your application. To understand what the application is doing and has, you have to know and understand the entire story. My feeling is that you need to minimize the surface area of the wiring configuration. As much as possible, I want the wiring information centralized into the smallest area of the code as possible.

As a negative example, take the idea of using declarative attributes. It's easy to write them, but where's the important information now? Everywhere. All over the place. It's easy to create that behavior with the attributes, but how do you understand what the application as a whole is doing after the initial write? You can beat the problem by creating "visualizers" that analyze the code and tell you the entire "story" of the application. That's a fallback solution, and extra work that you can avoid.

My very strong preference is to use what I call a Registry (I'm not sure that this is quite the same usage of the pattern that Fowler calls a Registry. I'd be happy to get some feedback about this terminology). In StoryTeller I have a TreeView control on the left strip of the application that acts as like the solution explorer in Visual Studio. When one of the TreeNode's is clicked or right-clicked StoryTeller exposes menu items that will execute Commands (Jeremy's Law of Software Design: all systems want to have an ICommand interface). I found myself frequently adding new ICommand's to the context menus, and wanted a declarative way to just say that a particular TreeNode executes these ICommand's. After a couple iterations, I arrived at using a class called the MenuRegistry that I use to wire up the TreeView menus that exposes a very simple fluent interface to attach commands to the different types of TreeNode's. The code (as of 1/14/2008) that specifies the comamnd to menu wiring is shown below:

```
        protected override void configure()
        {
            registerTypes();
            AddCommand<AddExistingProjectCommand>().AppliesTo<HierarchyNode>();
            AddCommand<CreateNewProjectCommand>().AppliesTo<HierarchyNode>();
            AddQueuedCommand<RunAllTestsCommand>().AppliesTo<HierarchyNode>();
            AddCommand<ImportFitnesseCommand>().AppliesTo<HierarchyNode>();
            AddCommand<RemoveProjectCommand>().AppliesTo<SystemUnderTestNode>();
            AddCommand<EditProjectPropertiesCommand>().AppliesTo<SystemUnderTestNode>();
            AddQueuedCommand<ExecuteTestCommand>().AppliesTo<TestNode>();
            AddCommand<AddTestCommand>().AppliesTo<SuiteNode>();
            AddCommand<AddSuiteCommand>()
                .AppliesTo<SuiteNode>()
                .AppliesTo<SystemUnderTestNode>();
            AddCommand<AddSetupCommand>().AppliesTo<FragmentGroupNode>();
            AddCommand<AddTearDownCommand>().AppliesTo<FragmentGroupNode>();
            AddCommand<AddNamedFragmentCommand>().AppliesTo<FragmentGroupNode>();

            AddCommand<DeleteCommand>()
                .AppliesTo<SuiteNode>()
                .AppliesTo<TestNode>()
                .AppliesTo<FragmentNode>();
            AddQueuedCommand<BatchRunCommand>()
                .AppliesTo<SystemUnderTestNode>()
                .AppliesTo<SuiteNode>();
        }
```

If I want to know what's hooked up to the different TreeNode's I can simply go to this one class and scan one method that expresses the wiring information declaratively. I've put the "signal" that I care about in one place and tucked the dirty "noise" about how the menus get done out of the way.

There's a lot of mechanics underneath the covers to instantiate the exact ICommand, relate it to the TreeNode that launched it, and build up the Menu on the fly. I'll write a separate post on that if somebody wants me to, but the important point is that the "story" of what's happening in the menus is boiled down to a little piece of the code.

19 The Command Executor

There's a thread on the altdotnet list this morning about how to unit test background operations originating from a screen Presenter. I have a strategy for that in StoryTeller that I think has worked out quite well. Instead of creating a background worker or thread directly in a screen presenter, I run all background actions through what I call a "Command Executor." When a presenter or a menu command needs to run some sort of asynchronous command I have that class delegate to an instance of an ICommandExecutor interface. The interface for the ICommandExecutor in StoryTeller is shown below:

```
    public interface ICommandExecutor : IStartable
    {
        void Stop();
        void ExecuteCommandWithCallback(ICommand command, ICommand callback);
        void ExecuteCommand(ICommand command);
    }
```

You'll see that the ICommandExecutor takes in the inevitable ICommand interface because this was written in .Net 1.1 and ported upwards later. I think for my new project I'm going to something more like this to take advantage of lambda expressions instead of cluttering up the code with extraneous class definitions.

```
public interface ICommandExecutor
{
    void Execute(Action action);
    void ExecuteWithCallback(Action action, Action callback);
}
```

Now, why would you do this? Using a BackgroundWorker or a background thread isn't that hard, but:

- The threading code is noise code that distracts the reader from the real meaning of the code. The CommandExecutor pattern let's us change the code semantics inside the Presenter into merely a call to "run this in the background."
- It's potentially a lot of repetitive code to set up threads or bootstrap a BackgroundWorker. You have to remember to do the thread synchronization every single time. By using a Command Executor you can write the multi threaded code once and only once, including the work to synchronize threads (I use a SynchronizationContext in my CommandExecutor for the callbacks. No need for AOP black magic whatsoever).
- You do NOT want any important behavioral or business logic code to be coupled to the UI machinery. I see a lot of teams absolutely screw themselves over by embedding logic directly into the DoWork event of a BackgroundWorker, rendering that code

effectively impossible to reuse or unit test. The CommandExecutor will help push teams to separate the behavioral logic away from threading infrastructure to make that code both easier to test and reuse.

- The background threading is rougher in unit testing. Not impossible, but it does add some significant overhead to the unit testing effort.

Let's take the testing angle first. The original problem was how to be able to unit test the background operation. The easiest way is to simply turn the asynchronous behavior completely off by substituting in a synchronous command executor in the unit tests like this:

```
public class SynchronousCommandExecutor : ICommandExecutor
{
    public void Stop()
    {
        // no-op;
    }

    public void ExecuteCommandWithCallback(ICommand command, ICommand callback)
    {
        command.Execute();
        callback.Execute();
    }

    public void ExecuteCommand(ICommand command)
    {
        command.Execute();
    }

    public void Start()
    {
        // no-op;
    }
}
```

Now you can simply run your unit test and test the asynchronous behavior without having to setup ManualResetEvent's or other thread synchronization machinery. In StoryTeller I usually just test that the proper ICommand message was sent to the ICommandExecutor and call it good enough. I'll then turn around and test the ICommand.Execute() method in isolation in another unit test.

So far, I've found this pattern pretty well eliminates all of my unit testing problems with background operations. For integration testing one of your challenges for testing a user interface is asynchronous events. One of the solutions I've found to this problem is to route all background operations through some sort of CommandExecutor so that the test harness has a single place to watch in order to synchronize the test script with the user interface.

20 The Main Players

The IoC/DI tool of your choice

These tools are good for more than just crowbarring mock objects into your code. When you're talking about extensibility and modularity, an IoC tool is an easy way to wire up new Views, Presenters, and Services to your existing application. If you take a long look at the CAB, you'll see that a major portion of it is related to Dependency Injection and (ObjectBuilder) and service location (the WorkItems). One of the main reasons that I've always been dismissive of the CAB itself is how easy it is to roll your own framework wrapped around an existing IoC tool. You could build a composite application without an IoC tool, but the responsibilities that were assumed by an IoC tool are just accomplished by something else (or hard coded which would pretty well shoot down the idea of a composite application). Obviously I'm biased here, but I think that using an IoC tool is a huge shortcut in building any kind of extensible application. Assuming that that level of extensibility is actually justified of course;-)

Application Shell

The main form that contains everything else. The responsibility of the shell is to hold the main components of the user interface like menus, ribbon bars, and holders (panels, docking managers, etc.) for the screens that get activated later. This should be fairly thin, and there's some very real danger in the ApplicationShell becoming a catch all that collects way too many unrelated responsibilities.

ApplicationController

The Application Controller pattern identified by Martin Fowler is “A centralized point for handling screen navigation and the flow of an application.” The ApplicationController controls the screen activate lifecycle and the lifecycle of the main form. In most of my systems other screens are activated by calling methods on the ApplicationController. In smaller systems the ApplicationController would also control screen workflow. In larger systems I would break some of that responsibility out into separate collaborating classes.

I should point out that the ApplicationController is probably going to be a full blown subsystem in its own right. The Single Responsibility Principle will often drive you to split the ApplicationController into some smaller parts:

- **Screen Conductor** — Controls the activation and deactivation lifecycle of the screens within the application. Depending on the application, the conductor may be synchronizing the menu state of the shell, attaching views in the main panel or otherwise, and calling hook methods on the Presenter's to bootstrap the screen. It may also be just as important to deactivate a screen when it's made the inactive tab to stop timers. My first exposure to a Screen Conductor was an insurance application that was built with web style navigation. Anytime the user moved away from a screen we needed to check for “dirty” screens to give the user a chance to deal with unsaved work. On the other hand, we also had to check the entry into a requested screen to see if we could really open the screen based on pessimistic locking or permission rules. We pulled our a Layer SuperType for our Presenters for methods like CanLeave() and CanEnter(). The Screen Conductor would use these methods and others to manage screen navigation.
- **Screen Collection** — In applications that have some sort of tabbed or MDI display

of screens, you usually need to track what screens are active. The easiest example I can think of is a TradeCapture application I built last year. When a user opened a trade from either a search screen or by entering a trade id, the trade ticket window would popup in a separate window. It was important that there only be one window for each trade up at one time, so if the user happened to request the same trade that was already opened, we would just show the existing screen instead of opening a brand new screen. We used a Screen Collection class to track the open screens so that we could retrieve the active screen for an individual screen. I've seen this pop up on at least 3 separate occasions now.

- **Screen Subject** — I've only used this a couple times, but it's going into my regular rotation from now on. I've built several applications along the tabbed display motif now. In many of them there's a need to respond to a navigation request by first checking if the screen exists to either make the already open screen be the active tab or to create a brand new screen and make that the active tab. I've gone to the idea of a ScreenSubject object that can determine if it matches an open Presenter and knows how to create the new screen if it isn't already open. ScreenSubject works very closely with Screen Collection.

Others

- **Bootstrapper and Registry** – The IoC container and the ApplicationShell has to be constructed somewhere. I like to put all of the "bootstrapping" into a class called, wait for it, Bootstrapper. It's very useful to keep this separate from the ApplicationShell itself, both to keep the ApplicationShell cleaner and also to enabled integration testing scenarios that test partial application stacks. Bootstrapper is also responsible for spinning up any services that need to run throughout the application lifecycle (like caching or the event aggregator). The Registry is an optional piece that contains configuration for a specific module. It might help configure the IoC container, add more menu items to the shell, or attach new services.
- **Model, View, and Presenter** – I won't rehash these pieces here. See the rest of Build your own CAB for more information. There's about a hundred different ways to divide and organize these three things.
- **Service** – The most overloaded term in all of software development. Roughly put, it's the classes that you interact with that aren't one of the other things. I would strongly recommend that
- **Command** – Gang of Four Command objects. "Jeremy's first law of enterprisey systems" — Every system of sufficient complexity will have an ICommand interface. This subject deserves its own chapter/post.
- **Event Aggregator / Event Broker** – Here and here. I'm going to include at least three different flavors of this in my book.

- **CommandExecutor** – Also known as the Active Object. I like to use Command Executor to standardize the way my applications use background threading.

```
public interface IShippingScreen
   {
       string[] ShippingOptions { set; }
       string[] Vendors { set; }

       bool InsuranceEnabled { set; }
       bool SignatureEnabled { set; }

       string StateOrProvince { get; set;}
       string Vendor { get; set;}
       string ShippingOption { get; set;}
       bool PurchaseInsurance { get; set;}
       bool RequireSignature { get; set;}
       double Cost { get; set;}
   }
```