

EVOLVING A .NET 3.5 APPLICATION

PURPOSE

In this paper, I'm going to outline some of the thought processes, decisions and issues we had to face when evolving a Microsoft .NET application from 1.x/2.x to 3.x. I'll look at how we helped our client to adopt the new technology, and yet still maintained a release schedule acceptable to the business.

You can have a look at a video of a portion of the software in action here:

http://www.ythos.net/article_images/ab2.wmv

I will avoid details of the proprietary aspects of the system, and I'm not providing any source code for the components we've built. On the other hand, I will cover in some depth the general architecture and design of the system, particularly as it relates to .NET3.x. As a bonus, there are quite a few code snippets and tips we picked up along the way.

INTRODUCTION

Digital Healthcare produces a software platform for chronic disease management. It is deployed into both hospitals (e.g. John's Hopkins, Baltimore, MD) and primary care providers, and enables all the individuals and enterprises that participate in the long-term care of large cohorts of patients (including the patients themselves) to share information and collaborate, despite being in different trust domains, and with different specialties. To add to the complexity, we deal with huge volumes of image data associated with the patient's care journey, and integrate with a wide variety of third party instrumentation and information systems.

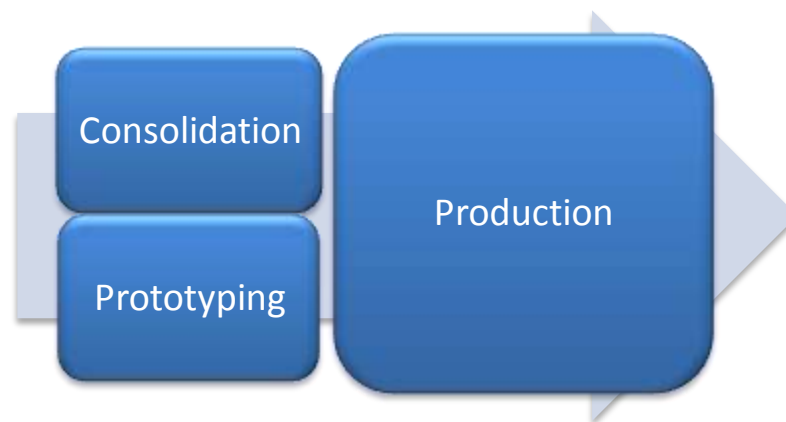
DH has been in the marketplace for over 5 years, and has recently been working on an evolution of our technology they've decided is sufficiently significant to call "V2". One aspect of the V2-ness is that components are to be built using the new .NET 3.x technologies like WCF, WPF and LINQ.

That decision was not taken lightly.

DH have 9 elapsed years and many millions of dollars of investment in a .NET V1/V2 platform, and the temptation is to squeeze every last ounce of juice out of that. There's plenty of headroom in classic ASMX/WSE web services, ASP.NET 2 web apps and Windows Forms. In the V1 app, the bulk of our client tier is Windows Forms for power users, with casual, occasional and "anytime, anyplace" users having an AJAX web interface to some of the overall feature set. Certainly, for the next year at least, they could rest on those foundations and happily extend our functionality, without hitting any serious brick walls.

That's not a very long-term view, though. Countless companies with great technology have gradually been rendered irrelevant by under-investing in the next generation of their products: they are leap-frogged by new entrants with more modern offerings, or find it increasingly difficult to integrate into a changing IT landscape (this, in particular, has driven change in our application). Equally, others have come to grief by embarking on a wholesale re-development of their applications – an undertaking which is universally acknowledged to take more time and cost more than any planning contingency could possibly imagine.

To attempt to limit the potential for failure, our strategy took a three-phase approach, with an iterative, incremental approach to change within those phases.



Consolidation involves some housekeeping on the existing platform to find and eliminate unused code, improve test coverage in areas that might be at risk of change, and replace code that depends on obsolete parts of the .NET framework. *Prototyping* is an investigative project to learn about the new technology, and how we might work it into the platform. *Production* is the build of new features and modules, using the new technologies.

We ran the consolidation and prototyping work streams roughly in parallel, but the “production” phase was dependent on a successful conclusion of the first two.

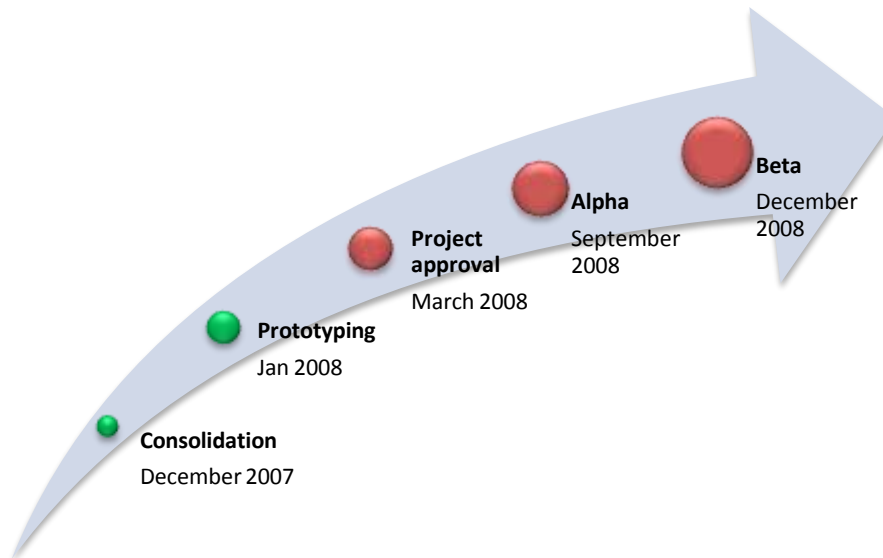
The consolidation project was potentially risky, as it is all too easy to disappear down a technology rabbit hole when you stop working on end-user scenarios to pay back a bit of technical debt. On the other hand, it is a mistake to ignore the compromises we know we’ve made to get something out of the door – especially when we’re about to start on some new work.

After an initial exploratory iteration, we set specific targets for the reorganization of the codebase, elimination of FxCop warnings and unused code, based on a “bang-for-the-buck” calculation as to what was worth doing, and what would just be pedantic. We also identified a few existing modules with the “black spot” for future replacement.

In the end, it consumed two people for about 3 months. That was about a month longer than originally estimated in elapsed time, but used 2 fewer people (who we stole for other projects), which was a reasonable compromise.

COMMISSIONING THE WORK

So, at the beginning of December 2007, we commissioned a project, gave it a codename containing a very bad domain specific joke, and set out a roadmap something like the following:



This is a snapshot from a report mid-January, which explains the green “started and all is well so far” blobs. Those blobs aren’t completion milestones (those appear on product plans, not the development roadmap). Instead, they’re “phase starts” – so we start prototyping in Jan 2008, we ship alpha product to early adopters in September 2008, and it releases through to a broader beta which starts in December. The roadmap acts as guidance to other departments (e.g. product management, sales) as to when we’re likely to need resource to help support the development process.

We also take decisions to ship specific components to early adopters way before the formal “whole product” alpha date. For example, we built a new ASP.NET 2.0-based compliance management tool and a new WCF-based async message bus component in early iterations, and they got shipped independently of the full platform.

Another thing to note is that formal “project approval” phase in March 2008. One danger with Agile methodologies is the flipside of its major strength: you only build what is needed right now.

The problem there is that you can find that your product is gradually, almost imperceptibly, iterating away from some overall business strategy, separately articulated, no matter how carefully you consider it in iteration/sprint/whatever-your-unit-of-work-might-be planning meetings.

We find it very useful to have a marker that gives you the opportunity to ask some deeper questions about the project. Are we building the right things – just not the things we thought we were going to need when we were strategizing? So should we reconsider the strategy? Or are we making short-termist decisions, and losing sight of our long term goals? Or is everything just peachy and we can go to the pub? Having a formal breather in the plan to lift your eyes up from the daily grind is very useful.

Having approved that roadmap, we started introducing the .NET 3.5 “Prototype” project into our work stream.

PROTOTYPING

Over a couple of months we built a working prototype of one portion of our overall application, that touched upon the key areas of the system – image and information visualization, remote working and messaging, reliability, performance and scalability.

There were a few important principles underlying this part of the project

- 1) We were never going to ship it in any way, shape or form
- 2) It didn't have to be "complete" – just complete enough to understand what issues we would face, and what the implications of various architectural and design decisions might be
- 3) Understanding means, in large part, "measuring" with realistic data sets
- 4) We would build it from "real" customer scenarios, with customer involvement, even though they were never going to have it delivered for production
- 5) The design decisions, compromises and brick-walls had to be well documented, communicated and discussed with the whole team, not just those involved in the prototyping, even if that is only through the medium of a few "whole team" meetings, PowerPoints and code sample sharing
- 6) We had some specific – but broad - objectives articulated on (e-)paper so that we, as a team, could ask and answer the right questions

Some of the questions we had in mind included

- 1) Can we reduce the amount of code we have to maintain? What sort of code can be eliminated?
- 2) What is the performance impact of an object-data mapping tier?
- 3) How long does it take to "think in LINQ/WPF/WCF"? For what types of developer?
- 4) How do we build tests around the new technology?
- 5) What are the integration points with our existing infrastructure?
- 6) What scaffolding do we need to build on top of frameworks before we can start building real applications

The actual questions to be answered are, of course, highly dependent on the nature of the project, but the basic principles: *ask specific questions, communicate your results, discuss and agree the implications* are a good set of guidelines for any prototyping exercise.

At the end of the prototyping phase, we had a clear understanding of the challenges and some of the benefits of the .NET 3.5 technologies, including raw performance measurements for things like LINQ-to-SQL abstractions of both our current database and various potential transforms and renormalizations of our database; what size and complexity of Visual Tree we could get away with for particular hardware configurations, how WCF channels of various kinds and configurations performed and could be configured in different deployment scenarios, and various different forms of message produced and consumed by our services.

We also had an idea of the learning curve in adopting the new technologies, and what kinds of things were useful to us.

The experience of going through this process has given us a top-10 list for "how to learn as a team".

LEARNING

- 1) Build stuff
Actually writing real code that addresses a real problem is invaluable; no amount of labs, samples or little spikes will show you where the road-blocks are.
- 2) Pair
Even if you don't normally pair, pairing while exploring a new technology is an excellent way of ensuring that knowledge is disseminated and assumptions challenged
- 3) Don't pair
Even if you normally pair, spend some quality time on your own just thinking, and assimilating information. Pairing is a pretty intense experience, and you'll need some breathing space

- 4) Get a good book...or two...or three; and read them; and use them
We bought a wide range of titles on .NET 3.x technologies. If you're buying for a team, get one copy of pretty much everything (for the shelf), and several copies of the ones you like the best (one per desk seems like a good number). There was universal acclaim amongst the team for [Programming WPF \(Griffiths & Sells\)](#) which I would echo (the fact that I've written about Windows Forms with Ian Griffiths does not influence my judgment – honestly!). We also liked [Programming Microsoft LINQ \(Pialorsi & Russo\)](#); it offered a good introductory overview. None of the WCF books were totally blow-you-away fantastic, but [Programming WCF Services \(Lowy\)](#) is pretty good. The WCF titles tended to be a little “cookbook-ish”, and not delve sufficiently deeply into the technology to really grok what was going on. Don't forget, though, that the book is only going to do any good if you read it. Knowledge doesn't magically get transferred by having it on the shelf. And it will only *really* make a difference if you work with it, build on the examples and refer back to it when you think you've understood something.
- 5) There are lots of ways of doing things “right”
Only some are more right than others. Make sure you understand what compromises you are making, and when that might be important. Always think hard about the choices you are making, but don't let indecision cripple your design. Make a choice, document the choice and explore its implications.
- 6) What are others doing?
Google (or, rather, Windows Live Search) is your friend. Someone has done what you are trying to do already, and they will have fallen into many holes along the way. Nodal blogs like Brad Abrams (<http://blogs.msdn.com/brada/default.aspx>), Mike Taulty (http://mtaulty.com/CommunityServer/blogs/mike_taultys_blog/default.aspx), Tim Sneath (<http://blogs.msdn.com/tims/default.aspx>) and Nicholas Allen (<http://blogs.msdn.com/drnick/>) are invaluable sources of information, both for their own writing and what they aggregate from elsewhere. If you're in the .NET 3.x world, you can't afford not to read (is that enough negatives to make a positive?) Scott Guthrie's blog (<http://weblogs.asp.net/scottgu/default.aspx>)
- 7) People learn in different ways, and at different rates
Never underestimate that what works for you might not work for other members of the team, and that everyone learns at different rates. You need to plan for that if you are going to introduce the new technology successfully.
- 8) Don't be afraid of paying for training, but don't pay too soon
Training is expensive, but a good way of getting access to real experts. I feel obligated to plug Pluralsight (<http://www.pluralsight.com>), and their upcoming web-based training (<http://www.pluralsight.com/main/olt/courses.aspx>), at this stage, but other training is available...
...not least of which is the stuff available to you through Microsoft – much of which is free, or pretty cost effective, often delivered or developed by the same experts. Regionally, you can find local events at <http://msdn.microsoft.com/events>, and there are usually links to online labs, screen casts etc. up there.
- 9) Don't be afraid of hiring in help
Although it didn't apply to us in this case, a contractor can bring in an enormous amount of expertise, and really bootstrap the project, especially in the early stages. Be very, very careful that you've road tested the consultant in some way, though, as the cost/benefit ratios can be a bit precarious. See them talk, get some good word-of-mouth – don't just go and ask for “an expert” from your friendly recruitment consultant.
- 10) Learn to use the canonical tools

There are some things that you really can't do without. If you had to pick three for your new-technology-development-desert-island, how about:

Reflector (<http://www.aisto.com/roeder/dotnet/>)

LinqPad (<http://www.linqpad.net>)

SvcTraceViewer.exe (<http://msdn.microsoft.com/en-us/library/ms732023.aspx>)

(Discuss?)

11) And the bonus 11th item: Admit your mistakes

During the prototyping phase, most of what you do is make mistakes. Most of the benefit of a prototype is learning what you wouldn't do again. Don't get too wedded to the choices you've made, and don't be afraid of abandoning them.

PLANNING AND SETUP

We completed the prototyping and review phase by the end of March 2008 (on schedule; miracles do happen), and took a look at the product plan to determine what the best "new feature" candidate might be to introduce the new technology to the team. After a few rounds with the product manager we decided that a new clinic and appointment management module would be an excellent choice, for a number of reasons:

- 1) We had a well-elaborated scope for the project, that was replacing and extending a very old, much unloved piece of functionality in the V1 application
- 2) It offered some compelling end-user value
- 3) It had well-defined points of contact with the rest of the platform
- 4) It would be a slice through the entire stack – data, services, presentation and integration

We put a team of 3 developers onto the project, one dev/test and about 10% of a PM, and estimated about 3 elapsed months to beta, with a May kickoff (when existing commitments were concluded). This included a considerable amount of contingency for the learning curve, but was also well ahead of the roadmap we plotted out in January.

We do our high-level planning around a 5 page document from the PM. In this case, it outlines 3 "product objectives" (the key value props which justify the new development), and also looks at how those objectives map into the gaps in our existing offering, right down to the level of calling out details where we're looking for specific improvements.

We brainstormed those into 10 key success scenarios (with PM, development and QA), along with a dozen or so QoS requirements. Over the course of the project these were elaborated into ~120 development tasks.

For the last couple of years, we've used Team System to manage our development process. When we kick off a new development activity like this (particularly a high-risk one), we create a new project in Team System, and branch its source tree from our HEAD release project. When we reach a "ship" point on a project, we merge back from the branch to the head, and perform our regression and release testing. There are maybe 2 or 3 of these projects running at any one time, with the aim of integrating 1-3 months from inception. We try to avoid longer projects by breaking them up into smaller subprojects. It gives us good control over the stability of the release while avoiding huge fragmentation of the code base into a million, unmanageable branches.

Individual iterations on the project tend to be of the order of 2 weeks, with about 1-1.4 work items completed per project per day. Although we run nominal 2 week iterations, we expect show-and-tell sessions from the team every week. One of the biggest risks to any project is a team going dark and when the whole project lasts 3 months, two weeks is a very long time. We also find that formal iterations are not totally useful – tracking continuous overall backlog and throughput in Dev and Dev/Test, and a qualitative measure of progress from

the PM is more effective than coming to artificial fortnightly “end-points”. The iteration is more a handy reporting point for the rest of the management team, and a way of “granulating” time when we’re estimating.

As code is checked in, we use the TFS2008 continuous build system, and run automated code analysis and unit tests alongside our actual builds. This was fabulously easy to set up (certainly by comparison with 2005!); the simple integration between the tools means that anyone in the team can maintain the build without needing the arcane knowledge of how Nant, NUnit, Draco, our home grown report XSLT etc. etc. all hang together, as per the V1 build environment.

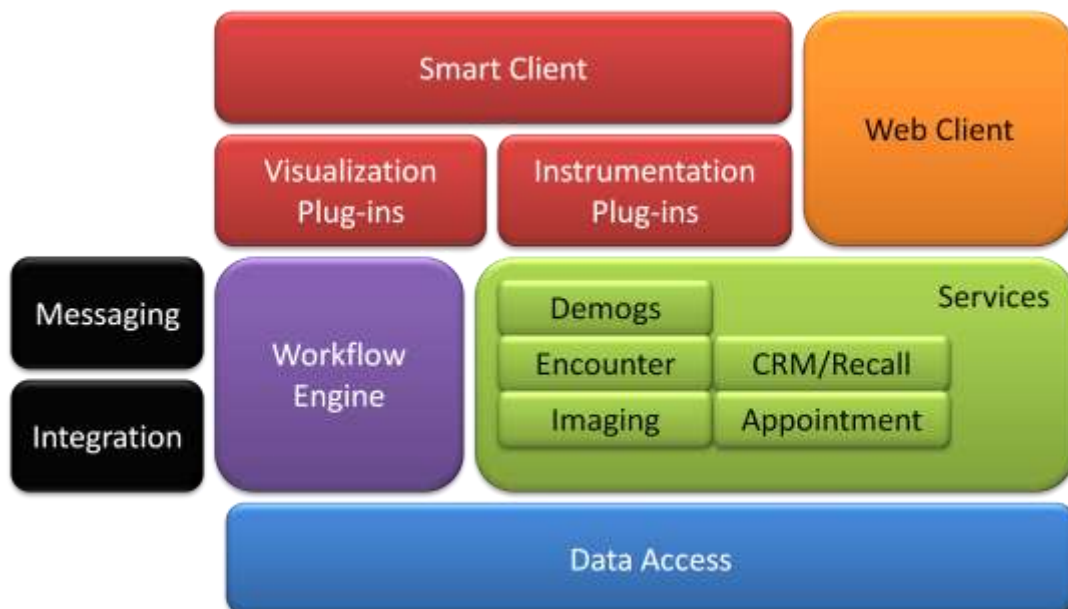
So...we’ve got a plan and a place to work... what now?

THE APPLICATION ARCHITECTURE - HISTORY

Our existing 1.0 application architecture consisted of a bunch of stateless services (note – not necessarily “web services”) that talk to a data access layer consisting of a combination of stored procedures and dynamic SQL. After much debate both internally, and with guidance from various folks inside MS, strongly typed data sets generated from our XML message schema were used to deliver information between the tiers – chiefly to facilitate Windows Forms data binding. Oh, how we regret this now. They are slow, heavyweight, difficult to serialize in all sorts of ways and runaway winners of the “most likely to break intellisense in the IDE” award from 2002-2007.

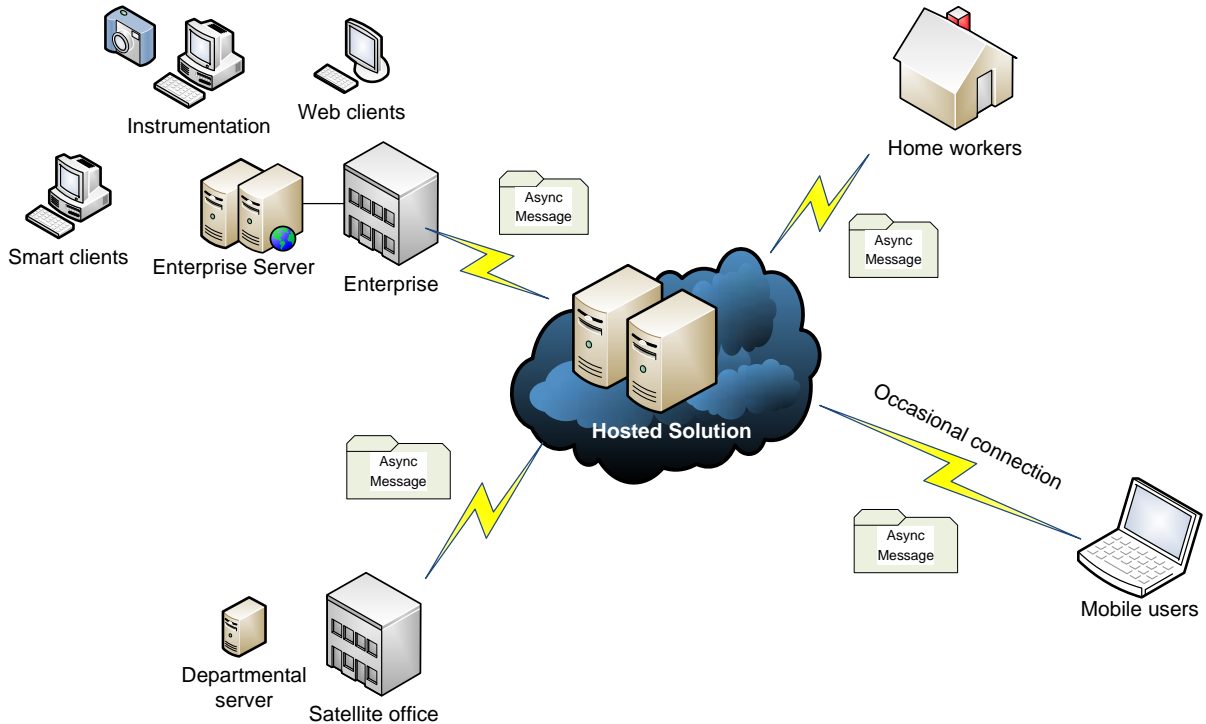
There are a bunch of middle-tier components that consume them - such as a workflow engine, an image storage and distribution engine, and a messaging and integration engine.

On the client side, a Windows Forms application shell discovers, hosts and presents plug-in components that are orchestrated to deliver the main application functionality. An *IServiceProvider*-based inversion-of-control mechanism allows the plug-ins to find and consume the services they need.



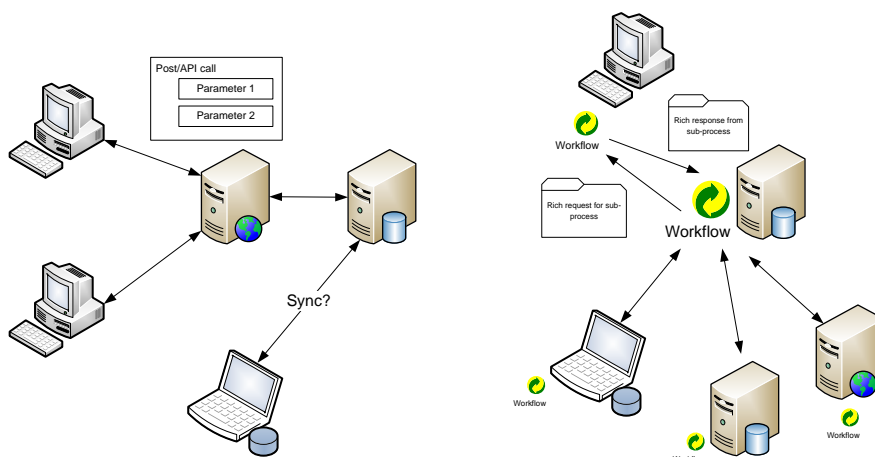
ARCHITECTURE BY BUSINESS PROCESS

You can see an example of the deployment architecture below. Notice how we support local enterprise-hosted versions of the application services that can then delegate out to services hosted in the cloud, to collaborate with home workers, mobile users and satellite offices.



To hook all this together, we use an asynchronous messaging pattern, orchestrated by our workflow engine. To understand what I mean by this, let's contrast with an n-tier system.

In the traditional n-tier system, clients request an action from the application server; the application server does some jiggery with its database (and any local cache) and returns some information based on the request. There are lots of ways to make that request: a post back from some form; a JavaScript call through to a web service; or an asynchronous call to a WS-* service and then a client-side wait until the information is returned, for example. However it is implemented, most n-tier applications are architected in this request/response kind of way.



Of course, this was a major step forward from the “open record set” architecture of days gone by where clients were permanently connected to the DB, and streaming through a set of results “live”, twiddling data as it went. And in terms of the *micro-architecture* of the application, this request/response mode of communication is just fine. But there are a few fundamental assumptions baked in to this view of the world that merit re-examination.

- 1) The application server can get to all the information it needs to do its job, and is the ultimate authority on that information
- 2) The client has access to all the data it needs to do any job it might have to do, typically via the application server, although possibly based on some local cache of what it has already seen / replicated from the server
- 3) The client can talk to the application server whenever it needs to
- 4) Somehow, everyone understands the implicit constraints on the order in which these (typically quite fine-grained) method calls should be made to keep everything jogging along correctly; or that there are no such constraints; or we deal with possible concurrency issues on a call-by-call basis.

To overcome these issues, we decompose the application not by the functions that are called between tiers (e.g. “get me the data for this”, “update the data for that”), but by the processes that are carried out at each location (e.g. “photograph me these patients today” or “this patient has rung up the call center and cancelled their appointment”).

You then design rich messages which contain all of the information absolutely necessary to carry out the process or sub-process at a particular location, and exchange those messages using the protocol of your choice. Each location manages its own local workflow to deal with the part(s) of the overall process for which it is responsible.

Because each message represents the start or end of a *whole (sub-) process* there is no need for the user to wait for a response – they can get on with their next job while the other end deals with the results of their labor. If there is a problem of some kind, then the resolution of that problem *is just another sub-process*.

From a *technology* point of view there’s very little difference between the two – they all end up sending data across the wire abstracted away behind the communications technology du jour (let’s say, a WCF-attributed method call). The difference is more philosophical; our message design has moved up the semantic food chain from the equivalent of the method call – one of many hundreds to complete a task, to a single (pair of) message(s) containing the request for, and final result of a whole sub process.

Although the success path is easy to deal with in this kind of architecture, failure cases make things more complex. As we have already mentioned, these failure cases have to be built into the workflow, too.

We see all the usual issues, including concurrency problems; transactionality; inexact entity identity resolution; occasional connection; and volumes of data that need caching at the endpoints for efficiency and/or offline use. We deal with them by looking at the overall process, treating the failure cases as just another interaction between sub-processes, and designing the workflows needed to deal with them.

The recognition of the intrinsically asynchronous nature of most of these kinds of processes leads to common design patterns such as queues of work for individuals or groups to complete, identification and recognition of the entities/artifacts that drive the workflow (are we tracking patients or parcels, or both perhaps?) and ownership and locality of those entities (who is “in charge” right now, physically and logically).

This “process”-based approach to information management is central to the successful adoption of an SOA – far more so than a particular deployment technology or application infrastructure.

Let's take occasional connection as our canonical example. As an industry, we commonly architect our applications as if we were "always connected", and then try to deal with occasional connection through synchronization and local caching. There are countless articles on the subject. This problem is "known hard", so there is a lot of plumbing people have built to help us. Such as the Microsoft Sync Framework <http://www.microsoft.com/downloads/details.aspx?FamilyId=C88BA2D1-CEF3-4149-B301-9B056E7FB1E6&displaylang=en>

That infrastructure is great, but it doesn't address the fundamental issue. Brian Noyes has had a talk for a number of years, the slide deck for which can be found here <http://www.softinsight.com/downloads/SmartClientOfflineDataCachingandSync.pdf> that discusses the problem in some depth – but the real meat of it (amidst a load of very valuable technical information) is in a bullet point in the summary section.

Design for disconnected operations early

Identify offline use cases

You should look at the *processes that are carried out at the occasionally connected endpoint*, model the information flows for those processes, and analyze the points at which information can and should be transferred.

This holds equally true for all parts of your application.

Design for your business process, not horizontal "platform-ish" stuff

Choose your infrastructure/platform based on your business processes

If you can't do that, you are necessarily compromising your business process; document and understand those compromises; understand your configuration options; share with the customer.

Understanding the ambiguities that emerge from that process-based view allows you to design process-based resolution for the issues, rather than data-based resolutions. It also makes it easier to identify how to insert human intelligence into the workflow, to disambiguate information at the right time, maximizing the benefits of automation elsewhere.

By moving away from "architecture by network and data" towards "architecture by business process" we have a means of communicating technical issues more clearly to our customers ("if these users can't see that server, you can't do this bit of the process at that time, in this way"), and we open the door to developing domain specific "solution design" meta-processes that are less tied to our technology, and more focused on customer needs.

A deep understanding of these kinds of process/workflow issues is fundamental to the success of our platform in our chosen implementation domains; but while these ideas are at the heart of our application architecture, implementing them is a whole different story!

IF I WERE YOU, I WOULDN'T HAVE STARTED FROM THERE

For our V1 product, this entire infrastructure was hand-rolled (and, indeed, we were still working out these kinds of message-exchange patterns in a theoretical kind of way). In Jan 2000 when Digital Healthcare was founded and we started development, .NET 1.0 hadn't yet shipped, even in Beta; BizTalk suffered the double

whammy of being not-quite-ready-for-primetime and brutally expensive. Web Services standards were in their infancy – W3C was just getting SOAP standardization underway, and the Bill & Steve Web Services Strategy announcement was still 6 months away. Windows 2000 was shipping, but not ubiquitous – most people were running some form of Windows 98 on the desktop, with NT4 as the dominant Windows server platform. I remember a long debate about whether to go with IBM DB2 (who he?) or this new SQL Server 2000 thing, which looked quite good on paper (so good, in fact, that it is only now, 8 years later, that we have dropped support for it – and that’s only to push a few laggards towards SQL2k5 so we can take advantage of its new features).

This ZD Net article by Mary Jo Foley from mid-2000 captures the zeitgeist pretty well – characterizing what is really the .NET framework/managed code strategy as “adding xml” to Microsoft’s existing product range: <http://news.zdnet.co.uk/software/0,1000000121,2079351,00.htm>

Even so, the period 2000-2002 was hugely productive for us. Our backgrounds were in C++/COM/ATL, and the productivity gains from moving to the managed environment were spectacular.

The transition to .NET 1.1 in 2003 was essentially seamless – it was effectively the bugs-shaken-out-and-ready for prime-time version of the .NET product, and it (not-coincidentally!) coincided with the public release of our new .NET-based product; 1.0 -> 1.1 was just a recompile for us.

Even the transition to .NET 2.0 in 2005/6 did not invite huge changes to our existing infrastructure. We exposed a few of our services through MS WSE, and took advantage of features like generics for new code. We also, for the first time, exposed some of our functionality through an ASP.NET 2.0 web application. In fact, the biggest driver for change at this time was actually getting it to build! Our highly-pluggable client tier had led to an (unnecessary) proliferation of assemblies to support a deployment granularity that was never needed, and the upgrade wizard choked horribly on our solution, ultimately consuming more GDI handles than the system allowed (a “bug” that is still in the product today). Where the 1.1 Visual Studio had been able to load and build the app (however slowly), VS2005 was deeply unhappy. But we *still* didn’t have to make any real code changes, or choices about the direction of the technology.

.NET 3.x was a different story. During 2004-2005 it became clear that the next generation technologies like Indigo (WCF) and Avalon (WPF) would be game-changing, and would require us to make some fundamental changes in thinking to make best use of the technology. After PDC2005, we started looking at it in some detail, and, during 2006, built some demonstrator components – basically some experiments with the data binding and graphics models, and some deep thinking about the right abstraction for the messaging and communications interface.

2007 saw us implement a couple of client components using WPF (a medical movie player with features like roving zoom and some image processing, and a clone of Vista’s calendar control), and re-exposing our image storage web service using WCF.

Comfort with that led to the decision to put both feet in the water and get a bit wetter; which brings us back to that big source repository and a bunch of scenarios to implement...

SMART CLIENT

During the prototyping, we didn't really start in one particular tier and make decisions in isolation – we looked at the problem in-the-round in order to see how choices made in the data tier, say, might influence decisions made at the client. However, we have to start the story somewhere, so I'll start with the client.

As I mentioned above, we couldn't possibly take the risk of building a whole new rich client app from the ground up in one go – we need to embrace, extend and replace our existing functionality piece by piece.

The logical conclusion from this is that there is going to be some kind of interoperability with Windows Forms required. Bring on choice number 1...

Do we host our new WPF components in the existing Windows Forms shell, or do we build a new WPF application shell and host our existing Windows Forms components in it?

This is not the totally trivial question it might seem. If you are in a Windows Forms host, then there are implications for WPF dialog ownership relationships, application level WPF resources and Application services. And there's an alter-ego set of problems for the Windows Forms components if they are hosted in a WPF world. Some of them are detailed here <http://blogs.msdn.com/scoberry/archive/2006/09/01/735844.aspx>.

This is also a very useful piece of knowledge if you are going to show dialogs in an interop world.

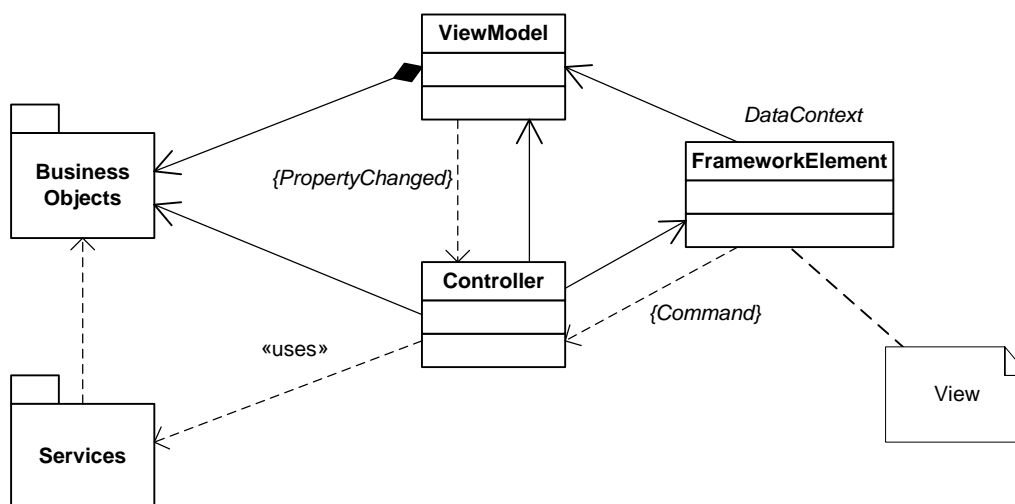
<http://blogs.windowsclient.net/rendle/archive/2008/03/13/wpf-windows-forms-interop-showdialog-extension-method.aspx>

(And it works backwards too; you can use *WindowInteropHelper* to set the *Owner* of a WPF dialog to a Windows Forms window handle.)

During the prototype stage we decided that it was going to be easier if we stuck to a Windows Forms application shell, and hosted our WPF UI within that. There are a couple of reasons for this. First, the Appointment Booking component we planned to build would take over the entire host when the workflow was launched by the user – essentially the AppShell was just providing window chrome at that stage. Second, some of the existing Windows Forms components really depend on their Windows Forms application object, and there's some low-level Win32 stuff going on. Although, at some point, it will be relatively straightforward to swap out those dependencies, one of our QoS requirements was to minimize impact on the existing codebase. We already had some things in mind that were going to have relatively far-reaching consequences, so we decided to play it safe in this particular area, and move on.

MODEL VIEW CONTROLLER/PRESENTER

The second fundamental question was about the presentation architecture itself. The particular variant of the MVC pattern that we hit upon looks like this:



So essentially, it is an example of the supervising controller/presenter pattern (<http://martinfowler.com/eaaDev/SupervisingPresenter.html>) which you see a lot in the WPF world.

The controller is responsible for “difficult stuff” – interactions with the services, maintaining complex model state.

The view is not entirely passive – it is data binding to the *ViewModel* (and occasionally doing just a little bit more than that, if it is all “view-state-ish” stuff).

The model is a big bag of information, capable of performing validation on itself, and providing some simple derived property logic.

Let’s look at the implementation of these in a bit more detail.

VIEW

The first thing to note is that the view has no special constraints – it is just a *FrameworkElement*; we opted for *FrameworkElement* rather than *UIElement*, *Visual* or *object*, because we want its *DataContext* property. That’s how we bind it to the *ViewModel*. We thought about abstracting this stuff away behind some *IView.DataContext*, but, frankly, that seemed like an unnecessary indirection – if we need to present some new kind of view, we’ve got another extensibility point for that (see below).

The view is pretty simple-minded. It is bound to a *ViewModel* by way of its *DataContext*, and it can respond to changes in that model, or update things in that model.

You know you’ve got your view about right if there’s pretty much no code behind, and what there is only deals with things like layout and animation.

While we’re on that subject, a few notes on views and visual complexity, and then a brief discussion of animation.

VISUAL COMPLEXITY

At the moment, there are a huge number of machines out there that are just on the edge of supporting WPF properly. When you are designing your visuals, try to implement something as simply as possible, using the simplest possible containers, and with as few visual elements as you can, to achieve the style you're looking for. Little tricks like using a *TextBlock* instead of a *Label* can save you vast numbers of elements in your tree.

There's a whole raft of useful performance notes in a document up on MSDN; it is well worth referring to this when you're doing a performance review. (And it is certainly worth doing performance reviews with the developers on a variety of different hardware, and documenting specific QoS requirements for your visuals)

<http://msdn.microsoft.com/en-us/library/aa970683.aspx>

ANIMATION

There are a number of useful tricks and tips when it comes to animation.

The most important thing is to use animation sparingly (but effectively). 400-600ms or so of subtle animation is extremely useful for covering up a round trip to the middle tier in some background operation, or indicating a transition from one process to another; don't expect them to be smooth as silk, though. A 3D rendering of the trench sequence from Star Wars where you substitute the exhaust vent at the end with your next modal dialog might look great on your Quad Core Dev box, but it is going to be awful on the 4 year old laptop with an Integrated Graphics processor. It may also, just possibly, violate some of your corporate style guidelines and present usability issues.

In addition to these high-level matters of principle, there are a few details we've discovered along the way.

Make sure that you've got the latest "known good" version of any graphics drivers on the target machines. Lots of weird bugs (like flickering, flashing and general misery) are down to the drivers.

Messing with the *RenderTarget* is much, much cheaper than messing with the *LayoutTransform*. Because it doesn't participate in the layout process, doing little zooms and rotates with the *RenderTarget* is a cheap and simple way of providing some visual feedback to the user as they interact with the system.

But even then, it can be too much like hard work for your machine. If you've got more than a few simple visuals in the tree under the element you're animating, don't try to animate it directly. When you want to start the animation, render the element to a bitmap, and replace it in the tree with an image displaying that bitmap.

Something like this:

```
Rect bounds = VisualTreeHelper.GetDescendantBounds(element);
RenderTargetBitmap renderBitmap =
    new RenderTargetBitmap(
        (int)bounds.Right,
        (int)bounds.Bottom,
        96.0, 96.0,
        PixelFormats.Pbgra32);

DrawingVisual dv = new DrawingVisual();

using (DrawingContext ctx = dv.RenderOpen())
{
    VisualBrush vb = new VisualBrush(element);
    ctx.DrawRectangle(vb, null, bounds);
}

renderBitmap.Render(dv);
```

```
Image image = new Image { Source = renderBitmap };
image.Stretch = Stretch.UniformToFill;
image.Width = renderBitmap.Width;
image.Height = renderBitmap.Height;
```

You can put the element back again once the animation has finished (if required). *Storyboard.Completed* is your friend. A little caveat, though – the *Storyboard* is not the sender of that event, the *ClockGroup* is. You’ll need a piece of code like this to retrieve the original storyboard.

```
Storyboard storyboard = (sender as ClockGroup).Timeline as Storyboard;
```

And you’ll need to give it a name, if you want to check which storyboard has just come back to you. You have to be a little careful about how you name it, if you want to avoid a runtime exception – the framework performs some validation to make sure it is safe to use as a field name in C#/VB.

Also, don’t be afraid of setting up and triggering your animations in the code behind. For animations based on size and position it is often easier to do that, rather than struggling with value converters and XAML. This is a good, general principal – WPF doesn’t mean XAML; if the code is more expressive, use the code. (For example, you might set up any event handlers you use in code, rather than mark-up, as the actual handlers themselves will be code too.)

Finally, I want to make a couple of observations about animation and render quality.

If you apply a *ScaleTransform* to an element containing text, then there is absolutely no guarantee it will ever look right again; this is a “feature” of the system which has been noted for a while, and is definitely better as of 3.5SP1, but still not fixed. You will be familiar with the fact that the text rendering is reduced in quality when animation is underway, but you may not realize that the engine sometimes fails to render text properly, even after it has switched back to “best quality” mode.

If you have significant amounts of text – especially if that text is rendered at small (<12pt) sizes – then using the bitmap animation technique is highly advisable, as it will allow you to avoid this problem.

In general, you will also want to ensure that you enable *SnapsToDevicePixels* for your UI. It makes the whole thing look crisper, at the expense of a small amount of positioning fidelity. It is inherited right down the tree, so you only need to set it at the top level of your UI (with a few caveats). There’s a post about it here.

http://www.thejoyofcode.com/Infinite_Resolution_and_SnapsToDevicePixels.aspx

Somewhat off-topic, but note that this inheritance doesn’t seem to work in Silverlight; you have to manually set it on every element.

DESIGN

Visual design is a vexed question for developers and designers alike. You only have to look at the shocking state of the web to realize that good UX designers are few and far between. Sometimes it looks slick, at the expense of usability (good use of whitespace in a harmonious layout is great; only being able to see 3 patients in a list of 100 because there’s loads of whitespace around them is not-so-great). Sometimes it looks like death on a plate, yet still conveys information really clearly. Sometimes it achieves a Tufte-esque ideal of both.

http://en.wikipedia.org/wiki/Edward_Tufte

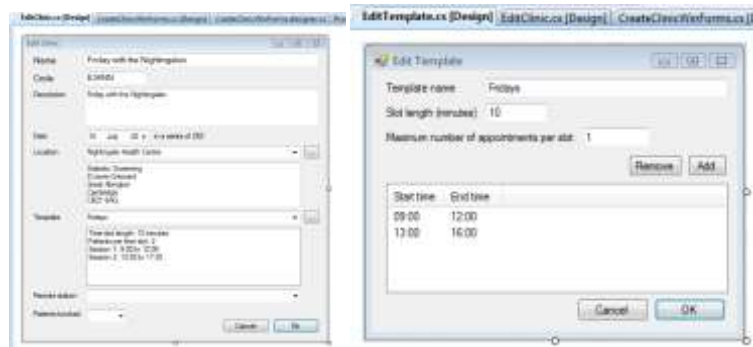
Getting developers and designers together to talk the same language is hard, and I can’t say we’re anywhere close to the answer.

However, there are some specific constraints we're working under because we are integrating into an existing app. Fundamentally, we couldn't make the new stuff look totally and utterly different from the rest of it. The contextual shock would have been too high, and users would have found it very difficult to use.

As it happens, we've just been through a corporate rebranding exercise. We have a new style guide, and that needed to be applied to the old app too, which gave us a little more scope for change. (That was one of the "consolidation" projects I mentioned at the beginning.) But basically, our buttons and layout had to chime with what was there already, and not stick out like a sore thumb.

Our plan is to gradually evolve the design over time, and bring the corporate websites, applications and other branding elements more and more tightly together. This is a start, but by no means the end, and educating our (freelance) graphics designers about what we need, and what the constraints are, is going to take time.

Initially, we worked out some scamps of the basic workflow and UI that make up a part of the user-experience on paper, and then a low-fidelity mockup in Windows Forms – no functionality beyond basic form-opening micro-workflows.



We use the Windows Forms designer rather than, say, Visio, because it gives a more "life-like" representation of the physical UI, and is marginally easier to add example data to lists. In the future, we might try using Cider or Blend for this bit, although one reason we picked the Windows Forms designer for the mockup is that there could be no temptation to "accidentally" use the mockup as the real thing!

We tried these out with customer representatives and iterated the design to the point where it was clear what the basic components would be, where they would appear on screen, and how we would work through the application workflow, as we started each of the common success scenarios.

We then built the entire application constrained by these basic frameworks, but without any styling or real thought to overall look, animation etc.

As we completed the basic logic of a component, we would then hand it over to someone else to style and perform the detailed layout and presentation. This could usually be achieved by changes to XAML and with reference to a common *ResourceDictionary*, but would occasionally require a return to the developer to tweak some code in the Views (particularly to help trigger more complex animations based on the existing layout), or to break out some embedded functionality into a sub-module.

This developer-to-designer workflow is enabled, fundamentally, by the UX designer and developer working together at the outset to get the workflow and basic layouts right. Everything else follows from that. If you have to make changes to the underlying layout or workflow, it requires considerably more cooperation between designer and developer.

I think we've all got a lot to learn in this regard. If I have one recommendation, it is to try everything with both new and existing users of your system, and get feedback as early and as often as possible.

That's more than enough about design. Let's get back to what views actually do.

VIEWS BIND TO MODELS

We chose to bind to objects, instead of, say, XML (for which there is a considerable overhead) or Datasets (no, no more, please! We've done our time...). In the section on the *ViewModel* below, you'll see how we go about that, with the aid of *INotifyPropertyChanged*.

We also make quite liberal use of *ValueConverters* to transform from the source type to something more suitable for the visual. Often this is simple *String.Format* stuff – and if you have the luxury of targeting .NET3.5 SP1 you can avoid the value converter completely with the new *StringFormat* property on the Binding.

One thing to be careful of, though, especially when using value converters: make sure that the binding is notified when *any* of the properties change that contributes to the final value of the binding.

Say I'm bound to a *Name* object that has two properties *FirstName* and *LastName*, and I've got a value converter that concatenates those properties to produce a pretty "LastName, FirstName" string to display.

All is well if the *whole name* changes (i.e. the reference is updated); but if just the *LastName* or *FirstName* changes, then you are in trouble.

There are a couple of ways around this. The first is to use a *MultiBinding* and an *IMultiValueConverter*, and bind to all the necessary components (root element, and each of the properties).

The other is to create a derived property on your class – say *FormattedName* and raise a property changed event for it when any of its constituent properties change – something like this.

```
private string _firstName;
public string FirstName
{
    get { return _firstName; }
    set
    {
        if (_firstName != value)
        {
            _firstName = value;
            RaisePropertyChanged("FirstName");
            RaisePropertyChanged("FormattedName");
        }
    }
}

private string _LastName;
public string LastName
{
    get { return _LastName; }
    set
    {
        if (_LastName != value)
        {
            _LastName = value;
            RaisePropertyChanged("LastName");
            RaisePropertyChanged("FormattedName");
        }
    }
}

public string FormattedName
{
    get
    {
        return string.Format("{0}, {1}", LastName, FirstName);
    }
}
```

```
} }  
}
```

There are advantages and disadvantages to both. The multibinding option means you don't pay for unnecessary *PropertyChanged* events that no-one is interested in, and you certainly avoid a proliferation of dozens of derived properties in your business objects for every type of formatting your UI needs.

On the other hand, some judicious "canonical format" derived properties encourage consistency in the application, and simplify the code you write in the client tier. They also make those canonical formats more easily available in other tiers – for logging purposes, perhaps.

VIEWS EXECUTE COMMANDS

Rather than raising and handling events in the old Windows Forms way, we have a new, higher-level abstraction – the command. Our views execute commands to request actions in response to user activity. These commands are typically handled by the view's controller, or some parent of that view's controller.

It is worth noting that we have opted to use regular common-or-garden WPF *RoutedCommands* for this.

This is by no means the only option.

The Microsoft WPF Composite Application Guidance (<http://msdn.microsoft.com/en-us/library/cc707819.aspx>), amongst other app frameworks, provides an alternative commanding abstraction – through a command broker of some kind, allowing various different pub/sub mechanisms for these kinds of events. Again, this offers a more sophisticated palette of options than regular commands.

Why do they do this? Chiefly, it is done to avoid having to couple the Controller to the View. WPF routed commands are handled by binding to something in the visual tree, and so you end up with some presentation code mixed in to the controller. The composite application guidance encourages an infrastructure that allows you to avoid this. A good explanation (and another approach to creating this kind of binding, using the WPF commanding infrastructure) is found in Josh Smith's CodeProject article <http://www.codeproject.com/KB/WPF/VMCommanding.aspx>.

In our infrastructure, the *Controller* is given an instance of its view in addition to its model, and it adds *CommandBindings* to that view's collection. It does mean that you cannot bind several views to the same controller (although they can, of course, share the same model, or be sub-components of the main view); should this be necessary, we would refactor to use a one-to-many abstraction.

There are obvious limitations to this; implementing "Save All" in a complex application, for instance, would appear to be difficult – how do we get all the peers to collaborate without something like the composite command from the composite app guidance?

Our solution to this is through composition support in the Controller itself, which we'll discuss below. First, though, a quick look at the ViewModel.

VIEWMODEL

The *ViewModel* aggregates all of the information needed for a particular controller to do its job. It implements *INotifyPropertyChanged*, and exposes its content as properties, in order to support the WPF data binding infrastructure, along with the binding validation mechanism.

That statement: “aggregates all the information needed for a particular controller to do its job” is very important indeed. ViewModels hold information from the services, shredded, aggregated and transformed in whatever ways are necessary to make them most palatable to the presentation layer (controller & view). They never, ever, get serialized across a service boundary.

VALIDATION

Validation is always a tricky problem. We’ve chosen to go down the route of implementing *IDataErrorInfo* on our entities, and specifying *ValidatesOnDataErrors=True* on the bindings in the UI. On the whole, we’ve also chosen to go for *UpdateSourceTrigger=PropertyChanged*, to ensure that you get validation-as-you-type behaviour. While this can have performance issues for very large numbers of bindings, we don’t see any problems with the typical number of active, editable things in our app, and it provides more immediate response to the user when there are errors. Your mileage may vary.

There’s a good article about *IDataErrorInfo* in WPF binding here:

http://www.thejoyofcode.com/New_in_WPF_3_5_2_Support_for_IDataErrorInfo.aspx

To make it easier to use, we’ve created a helper class to support the implementation of *IDataErrorInfo*, called, imaginatively enough, *DataErrorInfoHelper*. It has methods that let you get/set validators on particular properties in the Model, and then you call its *Validate()* and *ValidateAll()* methods from the relevant methods in the *IDataErrorInfo* interface. These validator objects implement an *IValidate* interface, and we’ve created a whole heap of useful validators (usually derived from an abstract *Validator<T>* base to help with some strong typing). Things like *DataRangeValidator*, *RequiredObjectValidator*, *RequiredStringValidator*, *StringLengthValidator*, *DataErrorValidator* (this last one effectively chains responsibility to some other object’s *IDataErrorInfo* implementation). They are simple to implement – you only need to override one method – and you can optionally set up a default error message.

Here’s the one from *StringLengthValidator* for your entertainment.

```
public override string Validate(string item)
{
    if (string.IsNullOrEmpty(item))
    {
        return ErrorMessage;
    }
    return string.Empty;
}
```

And here’s how we set up the validation for a person’s first name. Note that for stateless validators (like *RequiredStringValidator*) we’re using static singleton instances of the validator object. Lots and lots of validators get created, and it is easier on the working set this way. We might refactor things like *StringLengthValidator* in future to maintain a cache based on the length of the string, for the same reason.

```
SetValidator( "FirstName",
    new IValidator[]
    {
        new StringLengthValidator
        { MaxLength = NameStringPropertyLengths.Forename },
        RequiredStringValidator.Instance,
        NoLeadingTrailingWhiteSpaceValidator.Instance
    });
```

There’s another obvious future extensibility point here, in that these validators could be applied through configuration, allowing you to change some of these fairly low-level business rules without

recompilation/redeployment. We have another *IValidator* implementation that uses a simple rule language, loaded from external rule files to allow you to do this, if you need to.

Why did we roll our own *IValidator*-type infrastructure? We could have used something like the Validation Application Block from the MS Enterprise Library (<http://msdn.microsoft.com/en-us/library/cc309509.aspx>); or gone down a route more WPF-ish like this (<http://www.codeplex.com/wpfvalidation>).

Dealing with the second one first – we don't like this approach, because it inserts the validation rules into the View, rather than the business object model. It discourages sharing of rule sets between tiers, and that leads to all sorts of subtle but potentially fatal bugs.

As to why we didn't use the MS Enterprise Library block? We think it is pretty good; it does everything our validation framework does, and a bit more; it is moderately simple; it is going to be used (and therefore tested) by more people than will ever use our home-grown code. All compelling reasons to go that way. In fact, if you want to know how our infrastructure works, have a look at it. It is spookily (but coincidentally) similar.

However, we have much of this validation infrastructure already, in the Windows Forms app; it has been thoroughly tested in production (for robustness, functionality and performance), so there is no compelling reason to change, with all the risk that involves. If I was starting from scratch or dissatisfied with my own implementation, I would probably pick the Validation App Block, but as it is, we maintained the status quo.

VALIDATION DECORATORS – A CAVEAT

The visuals for the default validation are...ok... Nice and simple. You can't get much simpler than a rectangle, except, perhaps a square; or a point. Or you *could* argue that a circle is simpler than a rectangle (stop! stop!)

If you don't like the red box, you can replace it with something else, using the *Validation.ErrorTemplate* attached property. (<http://msdn.microsoft.com/en-us/library/system.windows.controls.validation.errortemplate.aspx>)

We haven't (at least, not in the video I showed you above; we've actually changed them to little red triangles in the corner of the control, now).

However, there is a problem – if the element is showing the validation adorer, because its binding is in error, and it gets hidden, or collapsed, then the adorer stays exactly where it was: unless you wrap the content (somewhere) in an *AdornerDecorator*. This is a notable problem for *TabControl*, but, in fact, we see it with lots of UI that can show, hide and generally get messed about with.

You can find some detail about this here:

<http://karlshifflett.wordpress.com/category/tips/>

The snag with this is that, once wrapped in the adorer decorator, it seems to participate in the control-tabbing order, and we can't find any reliable way of excluding it. This is a real pain, and if anyone has any insight into how to prevent it, I'd be very grateful.

COLLECTION VALIDATION – ANOTHER CAVEAT

A final note on validation relates to collections. We've chosen to use *ObservableCollection<T>* for our entity collections, as that supports WPF data binding very nicely, without too much overhead. We get notified when the collection changes, and the WPF binding happily calls our validation when items are added, removed and replaced. Our templates also bind happily to all the items in the collection, and our validation for those items gets called when they are changed.

But what if our validation *for the collection* depends on the values of the items in that collection?

For example, what if I have a collection of date ranges, which is invalid if any of those date ranges overlap?

Unfortunately, the WPF validation infrastructure doesn't bother to call on the validation for the collection when properties of items in the collection change.

To get around that issue, we've created a collection derived from *ObservableCollection<T>* called *ObservableContentCollection<T>*.

It overrides the *InsertItem*, *RemoveItem*, *SetItem* and *ClearItems* methods, to *Attach* and *Detach* from the *PropertyChanged* events. It then raises a new *ContentChanged* event should any element in the collection change.

We can attach to this *ContentChanged* event of any collection in our *Model*, and raise a *PropertyChanged* event on the model property itself, forcing the binding to revalidate.

We were extremely concerned about this approach – surely that would be a colossal overhead – effectively rebinding the entire collection whenever a child property changes? It turns out not to be nearly as bad as we expected. I don't know exactly how this plumbing operates internally, but it would appear that, while the validation gets triggered, WPF doesn't seem to do anything as nasty as rebuilding the entire visual tree from the templates if the actual entity itself hasn't changed.

So – that wraps it up for the model and the view. What about the controller?

CONTROLLER

We have tried a number of different approaches to defining the exact responsibilities of a controller, in such a way that it remains extensible, but easy to use and easy to implement for developers of all skill levels.

Rather than take a "free for all" approach, we've instead imposed some more rigid design constraints, to ensure that while it easy to use, it is also robust and (most importantly) inherently composable.

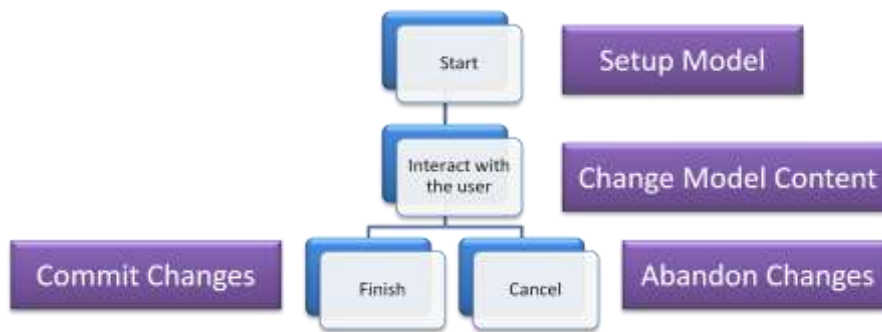
A controller embodies an atomic unit of work in the application.

It has a well defined life-cycle: Start, "operate", Finish/Cancel

It can retrieve more information at any time, but can only commit changes during "Finish"

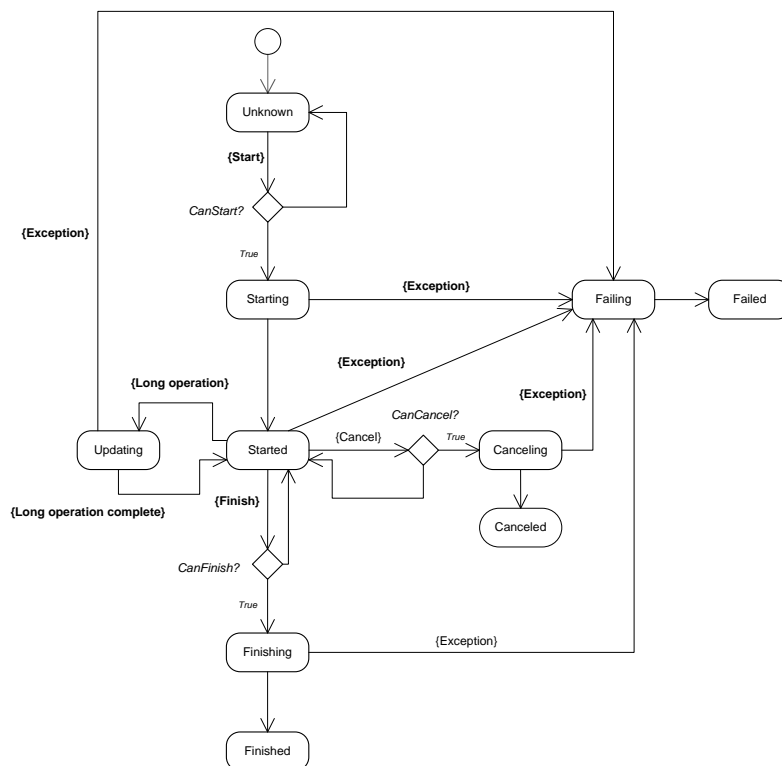
It also (optionally) has the ability to Finish (committing the operation) and then continue operating ("go do it again"), providing it maintains its model correctly under these circumstances

LIFECYCLE



Each of these major parts of the lifecycle has a corresponding “*CanXXX*” method – *CanStart*, *CanFinish* and *CanCancel*. This determines whether the controller is allowed to transition from its current state to its target state. The default implementation of these methods takes into account whether the model is in error (via its *IDataErrorInfo* implementation), and the current state of the controller.

A controller implements a fairly simple finite state machine something like this:



The state of the controller is exposed to the outside world (such as the view) through a *State* property in the model – this is an instance of an enum called *ViewModelState*.

```
public enum ViewModelState
{
    Unknown,
    Starting,
    Started,
    Finishing,
    Finished,
    Failing,
    Failed,
    Canceling,
    Canceled,
    Updating
}
```

The view can respond to the state of the controller changing by, say, presenting a long-operation indicator of some kind, showing error messages (in the case of *Failing* and *Failed*), or preventing the user from making changes while it is busy, or not yet started.

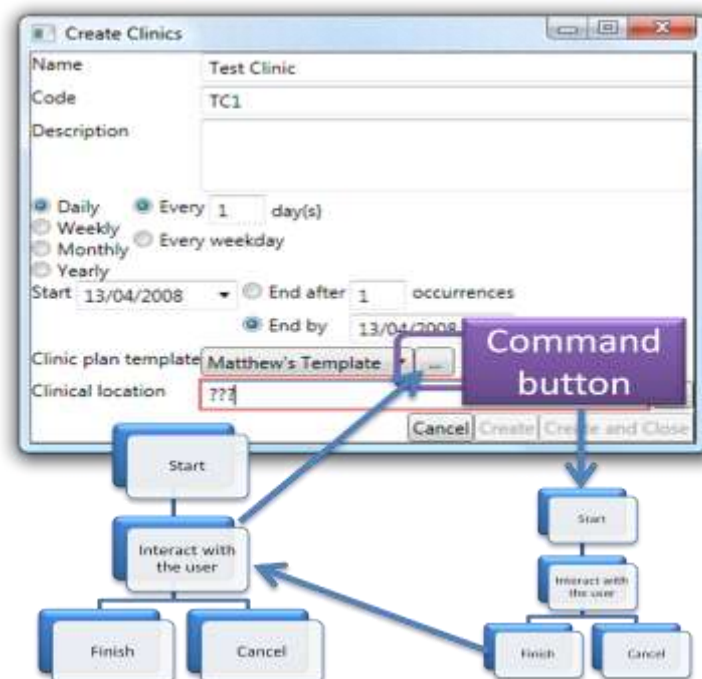
To perform your actual work, you override the various *Start*, *Finish* and *Cancel* methods in the controller itself, and add bindings for the commands that might be raised by the view. We'll talk more about the details of those overrides in [Long operations](#), below.

CONTROLLER COMPOSITION

There are two forms of controller composition: a *chain*, and *children*.

CHAINED CONTROLLERS

When a user clicks a button to request a particular action to be performed, you might startup a new controller (with a new piece of view) to handle the action. We call this a chained controller.

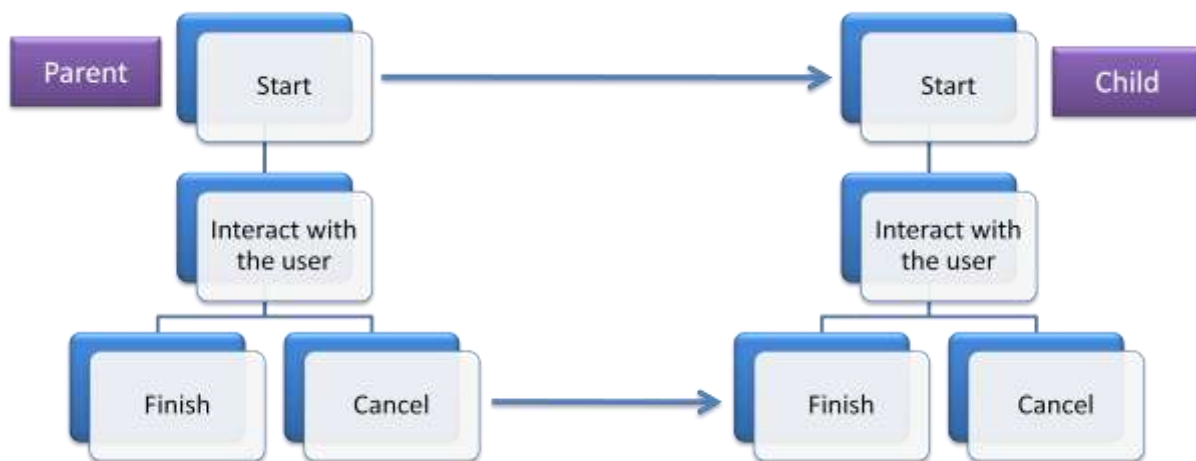


If the chained controller *CanStart*, then the command which initiates this new sub-process *CanExecute*. Executing the command shows some new UI, bound to a new model, and then calls *Start* on the chained controller. As this is such a common occurrence, we've built some infrastructure to handle these standard start/finish/cancel command patterns. We call them *Presenters*, and we'll discuss them in more detail in [Pulling it all together](#), below.

Chaining a controller does not necessarily imply a modal change – you can just as easily chain several controllers and show their UI modelessly. However, it is worth considering whether you really want a modeless view; if the controller deals with an atomic unit of work, it may actually be simpler to let the user deal with that work, then move back into their previous context. I guess the rule is – don't be a slave to either a modal or a modeless philosophy; think about what makes the job easiest for the user.

CHILD CONTROLLERS

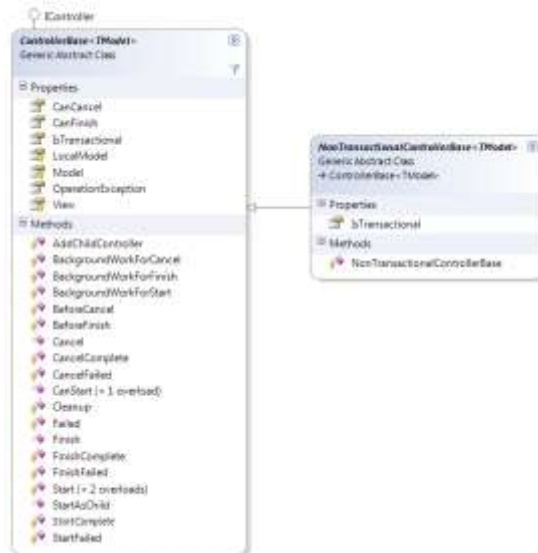
A controller also has support for *child controllers* – sub-controllers whose lifecycles are tied to that of the parent.



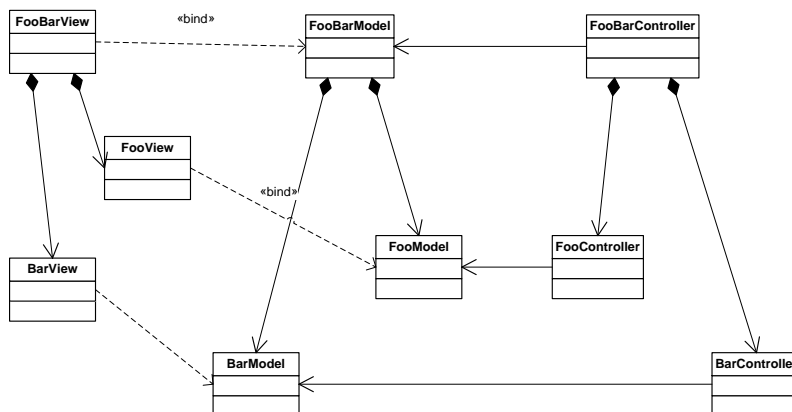
For example, you might create a re-usable component which consists of a box into which you can type a partial address (e.g. a zip code). As you type, it pops up a list of matching addresses, from which you can make a selection. This is being used as part of an "Update Patient Details" control, which has components for name, address, date of birth etc. So, when we start the *UpdatePatientDetailsController* we also want to start the *AddressSelectionController* (and similarly they need to be shut down together). In fact, the whole lifecycle for both needs to be tied together – what happens to the parent when a child fails to start, or fails to finish, for example?

After several iterations, we decided that when you start the parent controller, it starts itself, and then it starts its children. When you finish or cancel, it finishes/cancels its children, and then finishes/cancels itself.

It turns out that there are so many permutations of success and failure when you start to compose controllers that it quickly becomes a black art if you don't formally impose these constraints. So, we came up with our *ControllerBase* that implements the whole state machine for us. The controller implementations themselves need only concern themselves with their individual tasks.



So – if I have an application module called *FooBar* composed of *FooView* and *BarView*, with the *FooController* and the *BarController* doing their magic, I'd create a controller called *FooBarController*. Then I'd create a *FooBarViewModel* that aggregated the *FooModel* and the *BarModel*; and finally, a *FooBarView* containing the *FooView* and the *BarView*.



You can then add the Foo- and Bar- controllers to their parent. You do this by calling the *ControllerBase.AddChildController* method, which takes an instance of the controller (e.g. the *FooController*), the name of the property that represents its model (in this case the *FooBarModel.FooModel*), and the name of the child view element to which they should be bound. You'd typically do this in your override for *Start*.

When should you create a child controller? Typically, the answer is “not until you need it twice”.

We find that we frequently refactor model elements out into a sub-model, extract some piece of UI into a *UserControl* (for which Blend has convenient support), and extract some controller functionality into a child controller for reuse. Occasionally we do it to simplify the parent controller, if it starts to get over-large. These are, of course, the two classic reasons for a component-composition-based pattern of reuse.

Clearly, this pattern of composition is analogous to the composite command in the WPF composite application guidance referred to above, but with a built-in concept of a longer-running lifecycle, rather than a one-off “action”.

Although the example above is using a static composition model, you can easily extend this to a dynamic discovery and aggregation of the views, controllers and models. Our existing Windows Forms “review shell”

does something very similar right now – although it is more of an “active view” pattern, with a single, high-level controller building the shared model. We’ve shied away from that second-time-around to try to control some of the emergent complexity we discussed above.

LONG OPERATIONS

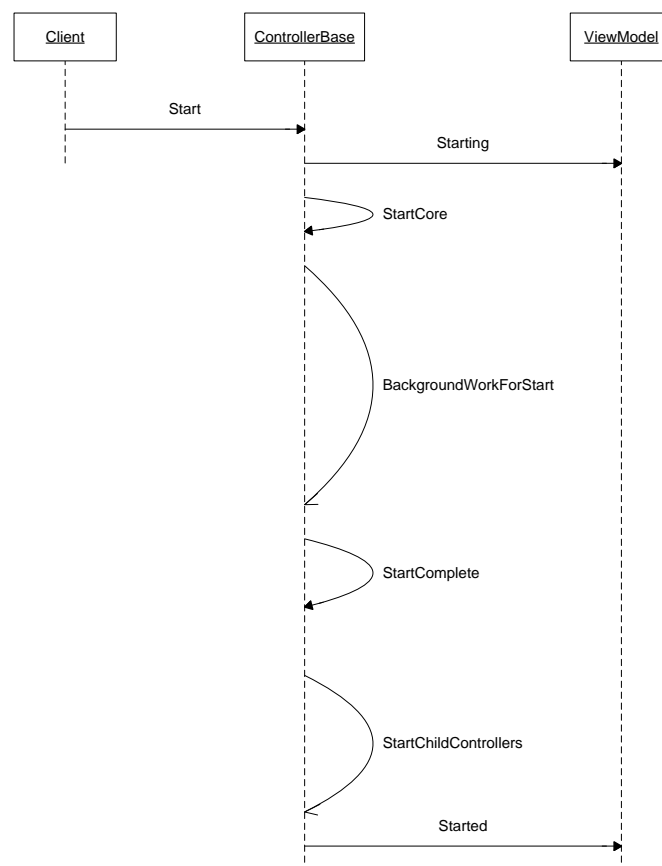
The bane of user experience is unresponsiveness. The web has taught us that people are prepared to wait for a remarkably long time (several seconds) for “full” content, as long as there is some indication that something is happening, and, preferably, there is some content of value with which they can interact while they are waiting.

<http://www.websiteoptimization.com/speed/tweak/psychology-web-performance/>

There are plenty of long ops a controller has to deal with: bootstrapping the model with information during *Start*, performing some form of search and display of additional information as the user interacts with the view, and committing changes during *Finish*.

To deal with background work during start, stop and cancel, *ControllerBase* makes use of the *BackgroundWorker* of .NET2.0 fame (<http://www.danielmoth.com/Blog/2008/05/backgroundworker.html>), and we’ve added a few constraints to help you avoid some of the pitfalls of working on a background thread.

To support this, *ControllerBase* breaks its operations up into several steps. Let’s take *Start* as an example.



When you build your own controller, you override *Start* to perform pre-prep on your model (and to add your own child controllers), *BackgroundWorkForStart* if you want to perform a long operation, and *StartComplete* to deal with the results of your long operation.

We considered passing delegates in for those methods, rather than overriding virtual methods, but decided against it. As we are expecting to derive from *ControllerBase* anyway, it seemed more natural. (We use the delegate passing mechanism elsewhere, though.)

In order to avoid disaster in your background worker, there are a couple of rules to be obeyed.

The View isn't allowed to update the model, or execute commands, if the model's state isn't "Started"

The Controller is allowed to read from the model in the background worker, but it is not allowed to make any modifications

Essentially, the state of the model arbitrates access to its content. Main and background threads don't collide because the model state (which is always updated on the main thread) enables them to avoid each other.

Controllers don't collide with each other during startup, because they are all looking at separate models. Parent controllers can copy data between models where they are shared to synchronize access during startup.

All updates to the model occur on the UI thread.

As mentioned above, in our V1 app we have a command-based composition mechanism where all our views share the same model all the time, and act on it based on user requests. This leads to combinatorially-complex behavior at runtime which is difficult to arbitrate and difficult to debug.

Using our model-and-controller-based composition model imposes an additional, hierarchical layer of management to arbitrate access to information – controllers are always owned by something (another controller) that knows how to arbitrate their access to the underlying model, especially in the face of multiple threads of execution.

HANDLING FAILURES IN THE PRESENCE OF ASYNC CHILD CONTROLLERS

As implied by the point about synchronization above, things are made considerably more complex by the presence of child controllers. Because child controllers can perform async work in their startup, *ControllerBase* hooks up to their state changes, then kicks them all off. It waits until they have all either succeeded and then changes our own state to "started".

If any failed, it cancels all those who succeeded, then cancels itself, and transitions itself into the Failed state.

And just because the world is the way it is, some of the children that started quite happily could still fail during cancel (think about the case where a disk goes away – permanently – part way through startup, for example). We take care of all of that in the base class, so that developers only have to worry about implementing *Start*, *BackgroundWorkForStart* and *StartComplete*.

A point for future consideration – perhaps *Cancel* should not be allowed to fail? It would simplify matters considerably, at little cost in terms of flexibility.

Whatever your state model is for the controller, you should define it, model it, and implement it in a base class. It is a *very difficult* problem to get right, and delegating it to every developer to deal with on their own terms is a recipe for bugs. Conversely, getting developers to think in terms of the state machine their controllers implement makes it easy to spot bugs early, and helps to get them thinking in terms of entry conditions – process - exit conditions for long-running processes, rather than individual actions.

A SIDEBAR ON THE EVOLUTION OF THIS MODEL

All of this is really about building a conceptual, and then practical application framework on top of the infrastructure we've got "in the box".

While you shouldn't make changes to your application framework lightly, it also shouldn't be seen as something that needs to be "done first" or even something that is "done".

Developers will come up with new infrastructural requirements all the time, and you should look at how the framework can be adapted and evolved to support them; or whether the developer should be discouraged from that approach, and look for another way.

For example, we didn't implement the child controller composition pattern in our first pass at the controller – it was added when we first needed it (which was fairly early on). We then extended to support the background work in *Start* and *Finish* – which was a *considerable* refactoring and extension job.

That is one good reason for making sure that you've got source code and a good understanding of the internals of any higher-level infrastructure components you do bring in from a third party. On the other hand, you also need to "go with the flow" of whatever components you do build/use. It is all too easy to try to bend a framework to your will, rather than going where it takes you.

OTHER LONG OPERATIONS - SEARCHING

Starting and Finishing is all very well, but what about long operations while the controller is running – what happens if I type "d" into my patient search field and hit the "Search" button?

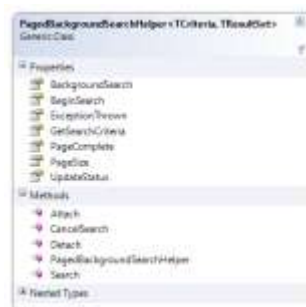
The view kindly executes the Search command, and the Controller handles it (not forgetting to set *e.Handled = true*, to prevent it routing on up to a parent...).

Unfortunately, going and finding all the patients with a "d" in their name might just take a little time, so we've built something to help us deal with this.

First, all of our "get lots of results" services implement paging (we'll take a look at how when we come to talk about the services, below). So, we have service APIs that looks something like this:

```
IEnumerable<ResultingThing> FindMeSomething(Some criteria, int pageIndex, int pageSize);
```

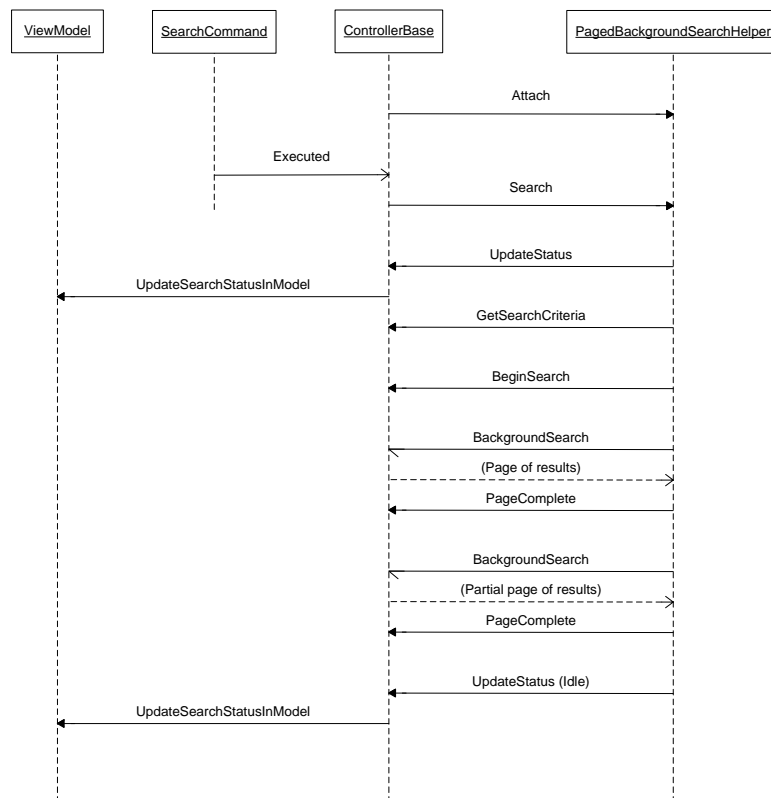
To make this easy to use, we've created a helper class called (and again, we're on an imaginative high-wire here) *PagedBackgroundSearchHelper*; it wraps up the use of background worker, specifically to deal with paging.



First, I apologize for the terrible *Attach* and *Detach* methods. These are basically *Startup* and *Shutdown* or *Open* and *Close*, or *Setup* and *Dispose*, or something similar. They'll change name. You call *Attach* some time before you start a search, and *Detach* when you're done searching: nothing too complex. Typically, you'll call *Attach* in *Controller.StartComplete* and *Detach* in *Controller.FinishComplete*, *Controller.CancelComplete* and *Controller.FailedComplete*.

The *PageSize* property determines how many results to return in a batch. The others are all delegates that you set to control the searching process.

The lifecycle of the paged searcher goes something like this:



Models for components which support these paged results typically provide a property called *SearchStatus* which is modified by the controller in the *UpdateStatus* callback. We provide an enum to capture the state:

```

public enum SearchStatus
{
    Idle,
    Updating,
    Searching,
    Canceling
}

```

Because it is common to provide “we couldn’t find any results UI” – but not before you’ve actually performed a search - we add a *HasSearched* (*HasSought?* *HasPerformedSearch?* We hate naming properties) property to the model, which we update from the setter on the *SearchStatus* property like this:

```

set
{
    if (!_searchStatus != value)

```

```

    {
        _searchStatus = value;
        if (_searchStatus != SearchStatus.Idle)
        {
            HasSearched = true;
        }
        RaisePropertyChanged("SearchStatus");
    }
}

```

Typically, we multibind the *Visibility* of the “Sorry mate, no results” UI to this property and the *Count* of items in some results collection, via a value converter.

This is such a useful pattern that we’ve created a derived *ViewModelBase* called *SearchViewModelBase* with these two properties in it.

In the video you can see a control that appears during searching that we call the *LongOperationEgg*. Its *Visibility* and a dependency property called its *BaseColor* are bound to the *SearchStatus* in the model via a couple of value converters.

Sometimes a long operation takes less time than you might expect. Because it might only be a few hundred milliseconds before the first batch of results comes back, you don’t necessarily want to pop up your progress UI (in our case, the egg) the instant a search starts. This is a great opportunity to use the system’s animation features to ease the UI in and out, in a less intrusive manner.

We use a trigger on the *Visibility* property to fade the egg in, and we don’t start fading it in for 250ms to give it a chance to complete without needing the UI. We could have specified a start time here, instead of the 3 key frames, but, you know – it was more complicated, once.

```

<ControlTemplate.Triggers>
  <Trigger Property="Visibility" Value="Visible" SourceName="OuterGrid">
    <Trigger.EnterActions>
      <BeginStoryboard>
        <Storyboard>
          <!-- Fade in over 1 second, after 0.25s -->
          <DoubleAnimationUsingKeyFrames
            Storyboard.TargetProperty="Opacity"
            Duration="0:0:1.25">
            <LinearDoubleKeyFrame KeyTime="0:0:0" Value="0" />
            <LinearDoubleKeyFrame KeyTime="0:0:0.25" Value="0" />
            <LinearDoubleKeyFrame KeyTime="0:0:1.25" Value="1" />
          </DoubleAnimationUsingKeyFrames>
        </Storyboard>
      </BeginStoryboard>
    </Trigger.EnterActions>
  </Trigger>
</ControlTemplate.Triggers>

```

OTHER LONG OPERATIONS – GETTING MORE INFORMATION

Another common pattern in our service implementations is the “get me some more information about this please” API.

For example, in the video above, we get some basic information about a patient (primary name and identifier) when we search, and then we get some more information about the patient (e.g. their current state in the active “care pathway”, details about their previous and next appointments).

We need to maintain a balance between making the roundtrip to the service too often, and returning too much information in one hit (to keep the message sizes sensible). Not forgetting that we need to keep the UI responsive while we do this.

Again, we leverage *BackgroundWorker*, but this time wrapping it with a queue of tasks that we're going to execute in the background.



It uses the same callback mechanism as our *PagedBackgroundSearchHelper*, and the same *Attach* and *Detach* methods to startup and shutdown (which, again, we call in the controller that is using it).

To add a unit of work to the queue, you call *AddUnitOfWork* and pass it an instance of a class derived from *UnitOfWorkBase*. You are warmly encouraged to (deep) copy the information you need to perform the work into an instance of this class, to avoid issues with access to your data cross-thread.

If the background worker wasn't already running, the helper starts some background processing (using the *BackgroundWorker* under the covers), which pulls items off the queue, and calls your *BackgroundWork* callback.

For example we might need to get some additional details about the history of the block of patients we've just found in our *PageComplete* callback from the search worker above. We define a couple of classes for our unit of work and the results

```

internal sealed class PatientAppointmentUpdateWork : UnitOfWorkBase
{
    public IEnumerable<PatientIdentity> Patients { get; set; }
}

internal sealed class PatientAppointmentUpdateResults
{
    public ReadOnlyCollection<PatientAppointmentBookingInformation>
        Information { get; set; }
}
  
```

Then we add the work to the queue for processing in our *PageComplete* callback from the paged search background worker.

```

// Make it update the availabilities
_backgroundPatientAppointmentBookingUpdate.AddUnitOfWork(
    new PatientAppointmentUpdateWork
    {
        Patients = patientLookup.Keys
    });
  
```

The unit of work is processed by our *BackgroundWork* callback (on a background thread)

```
private PatientAppointmentUpdateResults
    BackgroundWorkForPatientAppointmentBookingUpdate(
        PatientAppointmentUpdateWork unitOfWork)
{
    IAppointmentManagementService service =
        new AppointmentManagementServiceClient();

    ReadOnlyCollection<PatientAppointmentBookingInformation> information =
        service.GetPatientAppointmentBookingInformation(unitOfWork.Patients);

    return new PatientAppointmentUpdateResults
    {
        Information = information,
    };
}
```

And the results come through in our *UnitOfWorkComplete* callback (on the main thread).

```
private void UnitOfWorkCompleteForPatientAppointmentBookingUpdate(
    PatientAppointmentUpdateResults updateResults)
{
    foreach (var info in updateResults.Information)
    {
        LocalModel.Patients[info.PatientIdentity].AppointmentBookingInformation
            = info;
    }
}
```

The implementation arbitrates access to the collection for items added on one thread, and removed from another by using a lock. Incidentally, an excellent discussion on locking, timeouts and potential deadlocks can be found here: <http://www.interact-sw.co.uk/iangblog/2004/04/26/etmoretimedlocking>.

It turns out that clearing the queue mid-processing, in a way that is safe and restartable, is quite difficult. To manage this properly, we have a special unit of work called the *CancelUnit*. When you call the *CancelWork* method, the implementation enqueues an instance of the *CancelUnit*; the background worker recognizes this as special, and clears the queue for you, when it comes across it. From the client point of view, though, all you do is make that call to *CancelWork*.

When the queue runs dry, the background worker returns, shutting down background operations until the queue is filled again. Again, you have to be careful to hold the locks correctly to ensure that no one can be adding work as you're shutting down.

TRANSACTIONALITY

The remaining big problem for line-of-business apps comes under the general heading of transactionality. How do we support this Finish/Cancel model properly? How do we make sure that any edits we make to the model are rolled-back if we hit cancel, and committed if we hit finish?

There are two broad approaches to this

- 1) Copy-on-cancel
- 2) Copy-on-commit

The first approach allows you to edit the original object, but it (deep-enough) copies a snapshot before you start, and copies it back over the original if you choose to cancel.

The second approach makes a (deep-enough) copy of the original object, and allows you to edit that. It then uses the copy to commit the changes, and ensures that it copies whatever is needed back into the original object post-commit. Canceling comes for free.

The first approach is the one used by the *IEditableObject* pattern (<http://msdn.microsoft.com/en-us/library/system.componentmodel.ieditableobject.aspx>). In this world, your business objects become “live” and intrinsically support editability. WPF leverages *IEditableObject* (and *IEditableCollectionView*) as of .NET3.5SP1 (currently in Beta) for some controls (<http://blogs.msdn.com/vinsibal/archive/2008/05/20/wpf-3-5-sp1-feature-ieditablecollectionview.aspx>).

We use the second approach.

Our controller has a property *IsTransactional* which determines whether it supports this copy-on-commit behavior. Internally, a property called *LocalModel* which is the version of the model to which the view is bound, and *Model*, which is the original model passed in. If *IsTransactional* is *False*, then there is nothing for you to do. If it is *True*, then you are responsible for (deep) copying information from *Model* to *LocalModel* – you usually do this in *StartComplete*.

In *FinishComplete*, you are responsible for calling any services to enact your changes, and then copy back from *LocalModel* to *Model* any changes necessary to maintain the integrity of the model.

Why choose one approach over the other?

A justification for the first technique is that there is built in framework support for it. Another might be that you more often complete an edit than cancel, so paying the extra-copy penalty on the cancel cycle is cheaper.

There are a couple of significant reasons for choosing the second technique, though.

First, if there is a large amount of UI bound to your model, you may wish to operate in a sandbox, and avoiding any unnecessary updates in the rest of the UI until the change is actually committed.

Second, the sandbox makes it easier to limit the business rules that are in operation during the edit; you don’t get any unintended changes as the result of other parts of the UI interacting with the model as you modify it. This is particularly useful if you support modeless operation of some or all of your editors; a partial edit in one does not preclude a full edit in another, although it does impose an in-memory “last to commit wins” model, unless you add concurrency logic of your own.

TESTING THE CONTROLLER

The *ControllerBase* handles the lifecycle described above, and it is easy to write tests that validate that behavior. The only difficulty comes when the controller is performing operations that handle commands coming from the view. As it transpires, that’s not too tough either – you add tests for the private handler methods.

I’m not a big fan of testing private implementation. In fact, I think it is a very bad idea; in my opinion, tests should test public contract, unless there is something specifically algorithmic that, for good pragmatic reasons would cut down the amount of test code you needed to write elsewhere. This is one of the exception cases, though. Private methods that handle commands (or events) are de facto a part of your contract with the outside world, and most popular testing frameworks (NUnit, VSTS) support magical reflection-based private wrappers for exactly these circumstances.

INVERSION OF CONTROL

In our V1 application, we avoided dependencies on particular implementations of our services by implementing a simple inversion-of-control container which we exposed through an *IServiceProvider*-like container.

If you don't know much about this pattern, then Martin Fowler has a definitive discussion here: <http://www.martinfowler.com/articles/injection.html>.

Being able to substitute implementations of – say – our patient finding service with one that talks to a 3rd party database, and aggregates the results with those found in our own database – without having to rebuild the client components, has been invaluable.

There are plenty of inversion of control mechanisms out there: from good old *IServiceProvider*, to simple factory implementations, to (parts of) frameworks like Castle Windsor (<http://www.castleproject.org/container/index.html>) and StructureMap (<http://sourceforge.net/projects/structuremap>).

Where do you want to use inversion of control in the controller? Most commonly, when you need to get hold of service (client) instances to do the actual operational work of your controller. Although occasionally you might wish to use it to get hold of a particular view for a controller, or similar.

We inherited a hand-rolled inversion of control container from our V1 application, so we're using that. If I hadn't got one, or was unhappy with what we've got, something lightweight like StructureMap or the MS Composite Application Dependency Injection container would be ideal.

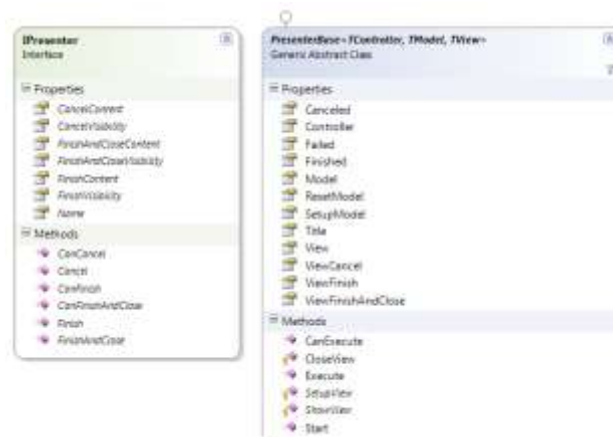
PULLING IT ALL TOGETHER

So we've built a model, view and controller for our module, and we're ready to present it to the world. How do we go about that? And how do we let the user interact with that Start/Finish/Cancel lifecycle?

We have wrapped that responsibility up into a class of entity called a *Presenter*. I know – there's a naming problem here; *Presenter* already has a more-or-less-well-defined meaning in the MVC/P/VM family of patterns.

What our *Presenter* does is to take a *Model*, a *View*, and a *Controller*, instantiate them all, bind to standard *Finish*, *FinishAndClose* and *Cancel* commands, handle the state changes (e.g. when your controller goes into *Failed*), and give you a means of bootstrapping them all and showing the UI in some way.

We've chosen to define a low-level *IPresenter* interface, and then a *PresenterBase* class that does most of the heavy lifting for you. They look something like this:



The presenter exposes *Execute* and *CanExecute* methods that can be plumbed directly into the *Execute* and *CanExecute* callbacks from a *RoutedCommand*. This makes it very easy to chain the execution of a new module from an action a user requests in a view.

The *CanExecute* method (by default) determines whether the command can be executed by the state of the model provided to it provided by a *SetupModel* callback. This allows the parent controller to provide an instance of the relevant model populated with all necessary information for the new controller to do its job.

A little wrinkle here is that it is important that the model shouldn't change between *CanExecute* and *Execute*, so our *PresenterBase* implementation actually caches the model it acquired in *CanExecute* for use in *Execute*. It also saves doing a load of potentially expensive work twice – once to work out whether you can execute the command, and once to actually execute it.

To implement a particular presenter, you override the protected *SetupView* and *ShowView* methods. *SetupView* allows the derived class to manipulate the view in some way before showing it. The presenter infrastructure actually instantiates the view for you, and sets its *DataContext* to the model; this is an opportunity to do some additional setup if required. *ShowView*, on the other hand, is where you actually display the view by some means. You can also override *CloseView* to remove the view from the display.

We've implemented a bunch of standard presenters – *RevealPresenter* (hosts the view in a *Reveal* element from the Bag'O'Tricks <http://i832.com/bagotricks/>), *DialogPresenter* (hosts the view in a popup dialog) and *PopupPresenter* (hosts the UI like a dialog in a layer within the main app workspace).

The important thing is that it doesn't actually matter all that much to our parent controller *how* the presenter is going to host the UI (as long as it is capable of providing whatever prerequisites are necessary in terms of the host view). We can easily switch between a modal dialog, for instance, and a modeless "roll-out", just by changing the type of presenter we use to present the module. Similarly, the presenter itself doesn't care what the controller or views actually do, so long as they support our standard interfaces.

Over time, we also discovered that we may want to provide slightly more custom extensibility points for our presenters, and as such we've implemented a bunch of additional callbacks – exposing the child controller's state through the *Canceled*, *Failed* and *Finished* callbacks. We also offer *ViewCancel*, *ViewFinish* and *ViewFinishAndClose* callbacks, which give you the opportunity to present some decision-making UI, and then perhaps abandon the operation, before we actually ask the controller to do the relevant action (typically this is used for "are you sure?" support).

BREADCRUMBS

One other thing the *PresenterBase* gives us is support for a breadcrumbs control.

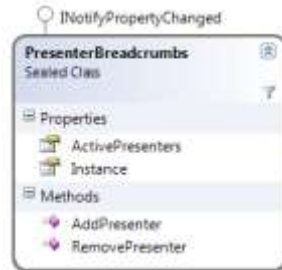
WPF natively has some support for breadcrumbs and a concept of back/forward navigation by using the journal provided by the *NavigationService* (<http://msdn.microsoft.com/en-us/library/system.windows.navigation.navigation.service.aspx>). It is very flexible, and can be used either with the existing *Page/Frame* WPF infrastructure, or through *CustomContentState*-derived items, overriding the *Replay* method to perform the relevant actions.

We didn't use it. Why not?

- 1) We aren't using the *Page* infrastructure, so we'd have to build our own custom state entities.
- 2) We do not require hierarchical navigation support (i.e. localized navigation within a particular navigation container).

- 3) The quality of documentation and number of implementation side-effects of the journal are somewhat off-putting

Basically, we wanted something simple that was easily plumbed into our commanding/controller/presenter infrastructure. We ended up with a singleton class (accessed through its *Instance* property)



There are two methods – *AddPresenter* and *RemovePresenter* that do exactly what you might imagine they do. There’s also a bindable, ordered collection of *ActivePresenters* that you can inspect for information about the current breadcrumbs trail.

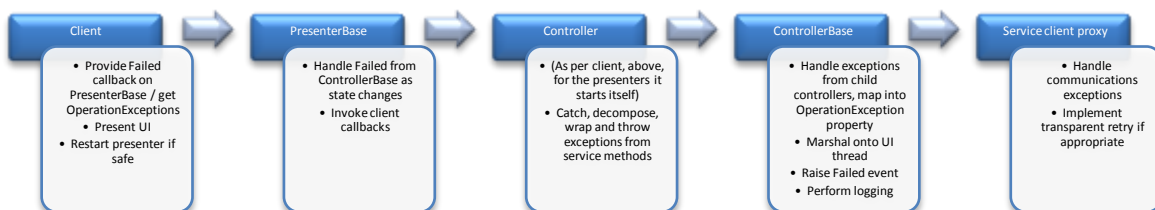
Our *PresenterBase* implementation automatically adds itself to the breadcrumbs trail when it is started, and removes itself when it is finished or canceled. Controllers, views, or even individual presenter implementations don’t need any knowledge of this to participate in the breadcrumb trail.

We provide a piece of UI that binds to the presenter instance, and displays the trail, along with some back, apply and finish buttons that map to the standard *Cancel*, *Finish* and *FinishAndClose* methods on the Presenter. (You can see this hosted at the top of our UI in the video).

EXCEPTION HANDLING

Handling exceptions during startup, shutdown and operation is a tricky business – especially when some of those operations can occur on a background thread.

It is important to define a clear exception-handling stack for your application, and to define the responsibilities at each layer. Ours looks like this



To deal with the background thread issue, we always marshal the exception onto the main thread, and transition to the *Failed* state (as per the state diagram above). We stash away any exception that occurred in the controller’s *OperationException* property, which exposes it to higher-level layers. This does not preclude access to the exception at the point that it was thrown (for debugging purposes), and we discourage re-throwing for all the usual reasons of violence to the call stack.

As we mentioned earlier, the *ControllerBase* implementation automatically cancels its child controllers, and puts itself into a “quiet” state (*Failed*) if it has a problem, but – unless your implementation does something explicitly to the contrary – it doesn’t touch the model. This should allow you to restart the controller in the

state it was before it failed (if you are dealing with a transient issue, for example). For debugging purposes, it also gives you the opportunity to inspect the state of the system as it was before it failed. This is another reason for using the copy-on-commit approach to model transactionality; we have both before-and-after information available to us after cancellation.

Because the rest of the system is known to be stable (or at least, hasn't yet failed), and exceptions do not percolate across controller boundaries, you have a much more robust recovery model than, say, allowing an exception to bubble all the way up to the top of the client tier. You might have completely abandoned and aborted the sub-process related to the controller that failed, but the overall workflow can continue.

LOGGING

In the client tier, we perform logging in our *ControllerBase*. The *ControllerBase* catches all exceptions that could be thrown by itself, its callbacks and overrides by derived classes, and manages the transition into the *Failed* state. It is therefore an ideal place to implement client-side logging, as it sees all the operational exceptions as they occur.

There are many logging solutions. For reasons of continuity, we picked log4net (<http://logging.apache.org/log4net/>).

SUMMARY

That was a pretty breakneck tour through our smart client components, and some of the design and implementation challenges we faced.

Our take-home story is that WPF does a lot of the basic lifting for you, in terms of a sophisticated data binding model and some great visuals and animation. However, that's not enough to provide a robust client application infrastructure.

One key feature you should be looking for in your client micro-architecture would be a separation of concerns between UI, interaction logic and business model rules, for maintainability and testability.

Above that, you will often need some kind of inversion-of-control mechanism to allow you to discover service implementations at runtime; we have always depended on this kind of mechanism, and while 99% of the time it isn't necessary, the 1% of the time when it is, you'll be glad you architected the application that way, as it is tedious to retrofit to a large codebase.

You should also be looking for a mechanism which allows you to compose the application from easily defined, testable modules, with well-defined boundaries of responsibility (although they do not necessarily need to be dynamically composed).

Finally, you need to deal with the big 3 problems for user experience

- 1) Long operations
- 2) Exception handling
- 3) Transactionality

We've attempted to address all of these issues with a Supervising Controller micro-architecture, and a base Controller class which directly supports composition, both in terms of child views and child controllers. We've layered as little infrastructure as possible on the View side, encouraging a "data binding only" approach, and we also encourage "the simplest visuals that could possibly work" to improve performance on older machines.

Finally, we've created a Presenter class which is responsible for bringing together an MVC triplet, showing the UI and binding it to the model. It also handles exceptions and state changes from the controller, and exposes a simplified interface over this for its clients.

BUSINESS MODEL AND SERVICES

INTRODUCTION

Our business model (which we implement in an assembly called *DigitalHealthcare.DataModelV2* for the sake of confusion) contains the entities which our services understand. They are the things which are serialized across service boundaries and sliced, diced and aggregated into our view models. They are capable of self-validation (for consistency, although particular services may layer additional constraints upon them).

Services define their contracts in terms of primitive types (like *int* and *string*), collections (*IEnumerable<T>*, *ObservableCollection<T>*) and entities from the business model.

It is not a straightforward decision to expose bindable, validated business objects from your service layer – you may have to think very carefully about versioning, and what the service contract might look like to non-WCF clients. That said - it is a bit of a no-brainer if you are not expecting to expose these services directly to other applications.

The granularity of services is also very important. We adopt this basic principle:

One service call should perform a complete, atomic operation

That doesn't mean that you can't aggregate other service calls behind that façade, but from the point of view of the client, they shouldn't be making multiple calls to complete one job.

There are a number of reasons for this, not least of which is related to transactionality. It is much easier (that is to say "more or less possible") to orchestrate your transaction from the middle tier, than it is to do so across a network hop.

This "higher level" abstraction naturally discourages CRUD (*Create Update Delete*) services, and pushes you towards an API that really does deliver a useful, reusable service which encapsulates some business value.

First, let's take a look at the business model in some more detail.

BUSINESS MODEL

We've already had a preview of our business model in the presentation section, above. Just as our ViewModel entities implement *INotifyPropertyChanged* to support data binding, and *IDataErrorInfo* to support validation, so do our business entities.

In fact, we implement an abstract base class called *EntityBase* to grease the wheels and all our business objects ultimately derive from that.

To help with validation, it uses our *DataErrorInfoHelper*, and exposes it through a property. It also provides an (implementation-free) virtual function called *SetupValidation* which you override to add your core validators.

```
DataErrorInfoHelper _dataErrorInfoHelper;
public DataErrorInfoHelper DataErrorInfoHelper
{
    get
    {
        if (_dataErrorInfoHelper == null)
        {
            _dataErrorInfoHelper = new DataErrorInfoHelper();
        }
    }
}
```

```

        SetupValidation();
    }
    return _dataErrorInfoHelper;
}
}

```

This property follows a common pattern for read-only properties of this type. We only construct the object on first use, and carry out any initialization at that point, too.

We use a similar pattern for collection properties. Here's an example from our Patient class, which holds the patient's names.

```

private NameCollection _names;
[DataMember]
public NameCollection Names
{
    get
    {
        if (_names == null)
        {
            Names = new NameCollection();
        }
        return _names;
    }
    private set
    {
        if (_names != value)
        {
            if (_names != null)
            {
                _names.CollectionChanged
                    -= new
NotifyCollectionChangedEventHandler(Names_CollectionChanged);
            }
            _names = value;
            if (_names != null)
            {
                _names.CollectionChanged
                    += new
NotifyCollectionChangedEventHandler(Names_CollectionChanged);
            }
            RaisePropertyChanged("Names");
        }
    }
}
}

```

SERIALIZATION

There are a bunch of things you have to do in your data model to ensure that it is all set for serialization. In addition to marking the class with `[DataContract]` (or `[CollectionDataContract]` for your collection classes) and the members with `[DataMember]`, you have to take a few extra precautions.

PROPERTY SETTERS

Note that we have provided a private setter for our read only property. If we left out the setter altogether, then `DataContractSerializer` would fail trying to deserialize the class. By making it private, we retain the benefits of a read-only property for the collection (i.e. you cannot simply assign another collection to it – not that the collection itself is read-only), but the WCF serialization will still work.

INITIALIZATION

Remember that no constructors get called on deserialization – not even your default ones.

If you have any post-deserialization initialization to do, then you have several options.

One is to implement some kind of *Initialize* method (rather like *InitializeComponent* in the Windows Forms world). That’s not very appealing although you could, in theory, deal with it in your client proxy.

Alternatively, you could try using the *OnDeserialized* attribute (<http://msdn.microsoft.com/en-us/library/system.runtime.serialization.ondeserializedattribute.aspx>). Personally, I don’t really like this as it adds an asymmetry between “normal” and “serialization” construction that I find to be a bit of a smell.

The final technique is to ensure that you don’t actually need any custom initialization at all – or, rather, any that *is* needed is performed “just in time”, inside method calls or property accessors. You can see that we are doing just that in the implementation of our *DataErrorInfoHelper* property; it initializes the internal validation on first use with a call to *SetupValidation*.

IDENTITY

Determining whether one instance of an object actually represents the same real-world entity as another is notoriously difficult problem, especially in a distributed system.

One cardinal rule applies, though:

Never expose a database integer id in your business layer.

This encourages people to forget about the distributed case, and has a tendency to tie your implementation to the particular database schema you’ve created, somewhere under the covers.

In our business object model, we attempt a marginally better abstraction by maintaining a *Uniqueld* for any entity of this kind (in the form of a *Guid*).

There may be several objects in play at any one time that have the same *Uniqueld* – each represents a different, independent view of the same underlying entity.

We considered building an entity cache in the client proxy that would allow us to resolve entities received as a result of any particular request to any instance of that same entity we might already have seen. However, we determined that this would be difficult (out of context, it is hard to work out what information should be copied from the “new” information to the “old” we had already cached), expensive of working set, and prone to errors. Instead, we leave it to the caller to re-integrate the results of a service call back into its own in-memory model.

However, it is important that the information returned from the service should maintain its own internal referential integrity (see under Service Contracts below).

To ensure consistency, we've created an *IdentifiableEntityBase* class which adds the *UniqueId* property. It derives from *EntityBase*, and is, in fact, the direct base class of most business objects in the system.



Of course, a *Guid* doesn't guarantee the identity of an object. If two *UniqueId*s are the same, then they definitely represent the same object, but if they are different, it does not mean that they must represent different objects.

Consider the case where two disconnected workers register the same patient. Locally, they generate *UniqueId*s for the patient – different unique ids, same patient. When they reconnect to the home server, and their requests to register a patient are messaged in, the fact that the unique ids differ merely means that they are *candidates* for addition – depending on who got there first, we may need to use other information to resolve the identity accurately (such as canonical identifiers, name and date of birth information, or even human intervention).

Designing service APIs that can support this accurately is a critical part of implementing a reliable distributed system.

ENTITY STATE

In addition to the *UniqueId*, *IdentifiableEntityBase* adds a few state-related items. *IsCommitted* is *false* if this is a new object that we haven't yet committed to the some kind of permanent store (via a service). *IsEdited* is a dirty-flag, usually set by client controllers if they edit the object. *IsPlaceholder* determines whether this object is being lazy loaded. If *true*, then it only has basic information (or no information at all), and further service calls will be required to furnish it with more information.

IsPlaceholder is a bit of a feeble property as things stand – it weakly describes the problem, not specifying exactly what has been loaded, and what hasn't; it is up to the controller to determine in context what it actually means. We had some debate about whether to go for a full lazy-loading model like NHibernate (<http://www.hibernate.org/343.html> - more on that when we get to the data access layer), and how tightly to tie our data access layer to our business model. In the end, we decided that simple is best, and we have not yet come up against a brick wall with this choice – although there are some occasionally difficult judgments to be made. Is something still a “placeholder” when it is has a lot of associated information, just not everything? Do we always end up having to track additional “loaded” state in individual controllers? It is quite likely that we'll kill this in future iterations, and maintain this meta-information in the view model.

SERVICE CONTRACTS

Our service contracts are implemented in a separate assembly called *DigitalHealthcare.ServiceContractsV2*. We directly share the implementation of these contracts between our client proxy, and our service implementations, rather than generating separate contracts in the client.

Having considered this very carefully, we've chosen to implement all of our service contracts through the path-of-least-resistance WCF approach of attributing-up a C# interface, along with our C# business model classes.

We could have defined our message schema by hand, and generated classes and interfaces, but that seems more trouble than it is worth.

```
[ServiceContract(Namespace =
    "http://www.digital-healthcare.com/ipServices/v2.0.0")]
public interface IPatientManagementService
{
    /// <summary>
    /// Get the primary details of a set of patients
    /// </summary>
    /// <param name="patientIdentities">
    /// The identities of the patients to retrieve</param>
    /// <returns>The collection of patients</returns>
    [OperationContract]
    [ReferencePreservingDataContractFormat]
    DigitalHealthcare.DataModel.PatientCollection GetBasicDetails(
        IEnumerable<PatientIdentity> patientIdentities);
}
```

VERSIONING

Note that we have chosen a simple versioning scheme, based on our namespace. (In fact, our first version was distinguished by having *no* version numbering in the namespace at all!)

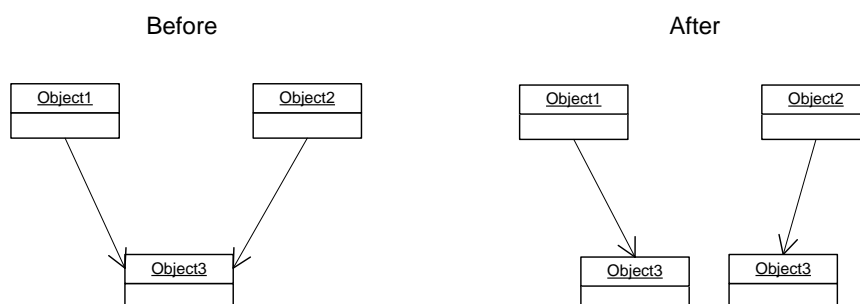
This version number is not the same as the application version, but instead, that of the service contract.

There are a number of options for versioning WCF service contracts, and there is an interesting discussion on the subject here.

<http://blogs.msdn.com/drnick/archive/2008/07/11/naming-contracts-for-versioning.aspx>

REFERENTIAL INTEGRITY

By default, *DataContractSerializer* does not maintain references in your object model. Any shared references become additional, unrelated copies of the same object.



What's particularly annoying is that *DataContractSerializer* actually supports a reference-maintaining implementation (making use of the xml *refid*), it is just a bit fiddly to use it.

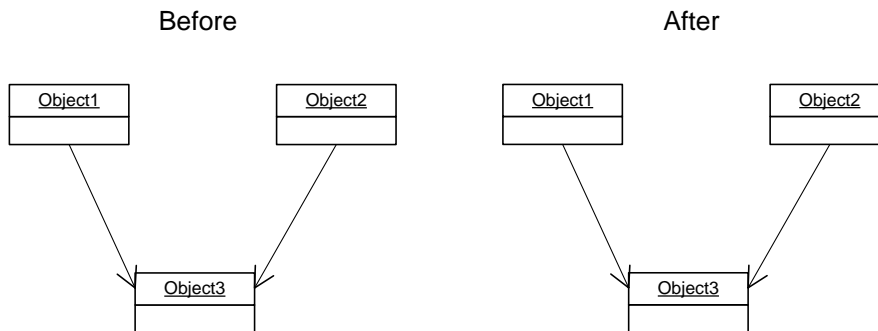
The canonical blog entry on the subject is here:

<http://blogs.msdn.com/sowmy/archive/2006/03/26/561188.aspx>

So, we mark all our data contracts with the *[ReferencePreservingDataContractFormat]*, and all is well.

Although there is a caveat in the comments about the code under .NET3.5 (<https://connect.microsoft.com/wcf/feedback/ViewFeedback.aspx?FeedbackID=307497&wa=wsignin1.0>), this only seems to apply to the imperative mechanism (i.e. calling *SetDataContractSerializerBehavior*).

The declarative (attribute-based) approach continues to work without modification, and it is this that we use in our code.



MESSAGE SIZE

When defining a service contract it is important to consider the likely size of your inbound and outbound messages. In the sections below we'll look at some techniques for limiting and/or minimizing those message sizes by configuring our client proxy or server, and our choice of protocols. However, the single biggest impact on message size is usually the design of the API itself.

Large messages usually fall into two categories: "big data", and "lots of data".

For big data, you have a couple of popular choices. You can, for example, define your service using a *Stream*, and then configure your protocol to use binary streaming or MTOM. <http://msdn.microsoft.com/en-us/library/ms733742.aspx> has an excellent article on the subject.

In our V1 application we define a service API that allows us to "chunk" large objects and send them in pieces, reliably restarting if transmission fails part way through the process. It sends and receives file chunks using a message like this one:

```
/// <summary>
/// This represents a chunk of a file for data transfer across the wire.
/// </summary>
[DataContract(Namespace =
    "http://www.digital-healthcare.com/FileTransport/schema/2007/1.0")]
public sealed class FileChunk
{
    private string _filename;
    /// <summary>
    /// The unique filename of the parent object
    /// </summary>
    [DataMember]
    public string Filename
    {
        get { return _filename; }
        set { _filename = value; }
    }

    private long _fileSize;
    /// <summary>
    /// The total length of the file in bytes
    /// </summary>
    /// <remarks>
```

```

    /// This is a zero-based index
    /// </remarks>
    [DataMember]
    public Long FileSize
    {
        get { return _fileSize; }
        set { _fileSize = value; }
    }

    private Long _chunkOffset;
    /// <summary>
    /// The byte offset of the chunk in the file
    /// </summary>
    /// <remarks>
    /// This is a zero-based index
    /// </remarks>
    [DataMember]
    public Long ChunkOffset
    {
        get { return _chunkOffset; }
        set { _chunkOffset = value; }
    }

    private byte[] _chunk;
    /// <summary>
    /// The portion of the file data
    /// represented by the chunk.
    /// </summary>
    /// <remarks>
    /// Note that this is a little-endian segment of
    /// the actual on-disk file stream. Most chunks will
    /// be exactly the requested chunk length. The last chunk,
    /// however, will not be padded beyond end-of-file, so may
    /// be shorter.
    /// </remarks>
    [DataMember]
    public byte[] Chunk
    {
        get
        {
            return _chunk;
        }
        set
        {
            _chunk = value;
        }
    }

    private byte[] _fileHashCode;
    /// <summary>
    /// A SHA1 hashcode of the entire file
    /// </summary>
    /// <remarks>
    /// This will only be filled in the last chunk of a file
    /// </remarks>
    [DataMember]
    public byte[] FileHashCode
    {
        get { return _fileHashCode; }
        set { _fileHashCode = value; }
    }

    public byte[] _chunkHashCode;
    /// <summary>
    /// A SHA1 hashcode of this chunk
    /// </summary>
    [DataMember]
    public byte[] ChunkHashCode
    {
        get { return _chunkHashCode; }
        set { _chunkHashCode = value; }
    }
}

```

We continue to use that technique in V2, although we could do this at the channel layer, rather than the client layer, and hide some of the implementation details away. There's some "getting started" information in this article on that option: <http://msdn.microsoft.com/en-us/library/aa717050.aspx>

For "lots of data", we prefer to use a paging technique we discussed above. If we have a lot of information to transfer from the client to the server, then we can use our "unit of work" technique to split the total job up into a number of pieces – although bear in mind that, unless we use heavyweight techniques (like WS-Transaction), we lose transactionality in that case.

If we have a lot of data to transfer from server to client, then we define a paged API (and use our *PagedBackgroundSearchHelper* on the client).

Implementing paging is very straightforward with LINQ.

```
var results =
    patients.OrderBy(
        p=>p.PatientId).Skip(
            pageSize * pageNumber).Take(pageSize);
```

Note the use of the *OrderBy* clause. It is imperative that you deterministically order your results, or the paging won't work. The magic is in the *Skip* and *Take* methods.

Note that one potential issue with any paging API is that it is possible for the underlying result set to change while paging is taking place (there is no transaction flow between requests for individual pages). Your client should therefore be prepared to receive a result twice.

SERVICE CLIENT

We hand-roll client proxies for our services, and publish them in an assembly called *DigitalHealthcare.ServiceClientV2* (which is referenced from our controllers).

This is one of the least-well explained areas of WCF, and merits a closer look.

```
public partial class PatientManagementServiceClient :
    System.ServiceModel.ClientBase<IPatientManagementService>,
    IPatientManagementService
{
    ...
}
```

The code above illustrates a simple client proxy. The *System.ServiceModel.ClientBase* automatically provides a channel for your service contract (as defined in the *IPatientManagementService* interface in the example above), and you are left to provide an implementation of that interface, which you do through your base class.

```
public DigitalHealthcare.DataModel.PatientDetails
    GetFullDetails(PatientIdentity patientIdentity)
{
    return base.Channel.GetFullDetails(patientIdentity);
}
```

When you construct an instance of the proxy, you must open it first, then call whatever service APIs you require, and then close it. Something like this:

```
try
{
    myProxy.Open();
```

```

        myProxy.GetPatientDemographics(patients, demographicKinds);
        myProxy.DoSomethingElse(patients);
    }
    finally
    {
        myProxy.Close();
    }
}

```

However, there are all sorts of issues that arise if the proxy faults during operation, and there is a recommended best practice for shutting down after a channel has been used. We have encapsulated that in a *CloseSafely* method that you can use instead of *Close*.

```

public static void CloseSafely(ICommunicationObject commObject)
{
    try
    {
        if (commObject.State != CommunicationState.Faulted)
        {
            commObject.Close();
        }
        else
        {
            commObject.Abort();
        }
    }
    catch (CommunicationException)
    {
        commObject.Abort();
    }
    catch (TimeoutException)
    {
        commObject.Abort();
    }
    catch
    {
        commObject.Abort();
        throw;
    }
}

```

I'm slightly tempted to turn this into an extension method for *ICommunicationObject*, but at the moment it is just a helper.

In order to discourage API designs that involve multiple service calls on the same service to achieve our ends, we actually wrap all of this opening and closing up into each service call in the client proxy itself, making it a "one-time" use. If you try to make a second call on the same proxy, it will throw an exception, because it has been shut down.

```

public PatientDetails GetFullDetails(PatientIdentity patientIdentity)
{
    try
    {
        Open();
        return base.Channel.GetFullDetails(patientIdentity);
    }
    finally
    {
        ServiceClientUtilities.CloseSafely(this);
    }
}

```

This is a somewhat unusual approach, but it does emphasize an important tenet of our architecture - and cause a runtime exception if it is violated. As time goes by, we may relax this restriction, even if it does not

become an issue – it is only a client-side decision and can be reversed in whole or in part without affecting the rest of the application. However, the straight-jacket is quite comforting at the moment.

MESSAGE SIZE AND UNNECESSARY INFORMATION

When you send an object graph across the wire to support a service, you can often be serializing and sending huge amounts of extraneous goop that is not required by the service, but that is possible to send because your object model supports it.

For example, a patient might have a name, an address, a date of birth, and a collection of encounters. If we have a *PatientManagement.UpdatePatientDemographics* service, its contract might well take a *Patient* entity to update. However, we wouldn't want to serialize their entire encounter history and send that across the wire too.

There are a couple of options. One is that we shred the patient into, say, a patient *Uniqueld*, the updated name, the updated address and the updated date of birth, and pass them as individual parameters to the API.

There are several places where we do exactly this. However, to help avoid the type-mismatch problems of passing *Guids* around, we use strongly-typed *Identity* objects (e.g. *PatientIdentity*, *EncounterIdentity*), which are, in fact, just *Guids* under the covers.



```
[DataContract(Namespace =
    "http://www.digital-healthcare.com/ipServices/V2.0.0")]
public sealed class PatientIdentity :
    Identity<Patient, PatientIdentity>
{}
```

In other cases, where it would be a considerable burden for the client to shred the information, we implement a “copy a subset and send” approach in the client proxy. This makes the solution transparent to callers, and, because the proxy is built by whoever builds the service contract, allows us to maintain a good correlation between what comes across the wire, and what is needed by the service.

For example, the demographic update API above looks something like this in the client proxy:

```
base.Channel.UpdatePatientDetails(
    CleanPatientToDemographics(patient),
    deletedIdentifiers,
    deletedNames,
    deletedAddresses,
    deletedTelephoneNumbers,
    deletedDemographics,
    LoggedInClinicianIdentity);
```

And the *CleanPatientToDemographics* method copies just what it needs for the service to do its job

```
private Patient CleanPatientToDemographics(Patient patient)
{
    Patient rc = new Patient
    {
        UniqueId = patient.UniqueId,
        IsCommitted = patient.IsCommitted,
        IsEdited = patient.IsEdited,
    };
    rc.Demographics.AddRange(patient.Demographics);
    rc.Addresses.AddRange(patient.Addresses);
    rc.TelephoneNumbers.AddRange(patient.TelephoneNumbers);
    rc.Identifiers.AddRange(patient.Identifiers);
    rc.Names.AddRange(patient.Names);
    rc.PrimaryAddress = patient.PrimaryAddress;
    rc.PrimaryIdentifier = patient.PrimaryIdentifier;
    rc.PrimaryName = patient.PrimaryName;
    rc.PrimaryTelephoneNumber = patient.PrimaryTelephoneNumber;

    return rc;
}
```

Choosing the right approach for your particular service contract requires some careful thought, balancing ease-of-use against message size and the overhead of copying.

HOSTING

We implement our services in an assembly called *DigitalHealthcare.ServicesV2*. We also provide a class called *WcfServiceProvider*, which implements static *Start* and *Stop* methods to host and shutdown all of the services we wish to host.

Each service is hosted using a code snippet like this

```
private static ServiceHost advancedSearchHost = null;

private static void StartAdvancedSearchHost()
{
    advancedSearchHost = new ServiceHost(
        typeof(DigitalHealthcare.Services.AdvancedSearchService));
    advancedSearchHost.Open();
}

private static void StopAdvancedSearchHost()
{
    if (advancedSearchHost.State != CommunicationState.Closed)
    {
        advancedSearchHost.Close();
    }
}
```

By implementing this separate class, we can choose to host in a Windows Service (for production, with all the advantages of auto-start and auto-restarting on failure); or in a command line application for debugging purposes (although, more often than not, we host in the standard WCF test host under the VS debugger).

Our standard implementation relies on the *app.config* to configure and initialize the service. A discussion of WCF service configuration is well beyond the scope of this document; but we tend to use two standard configurations:

- 1) *NetTcp* where we are operating inside a corporate network, and talking WCF->WCF
- 2) *wsHttp* where we are operating across the public internet, or where there are peculiar trust relationships

Bear in mind that NetTcp is capable of processing almost an order of magnitude more messages than wsHttp, so NetTcp is preferred, if you can use it. Message sizes also seem to be smaller with the binary encoding it uses. See Message Size below for more information on the implications of this.

SECURITY

In both cases, we will use some combination of transport security and/or message security, secured either by Windows identity, or a certificate (perhaps mapped to an internal windows identity through AD). As usual, Nicholas Allen has several great blog articles on the subject, including this one.

<http://blogs.msdn.com/drnick/archive/2006/04/14/configuring-http.aspx>

Another caveat to be aware of is the identity of the user under which the service is running on the server side. If it is one of the built in users (like Network Service) then all is well. If not, then you need to specify the *servicePrincipalName* in the identity specification on the client side (as well as on the server), or the service will not authenticate correctly. The identity is the qualified name of the user under which the service is running.

Essentially, this is allowing the client to authenticate the server, rather than vice-versa. It helps to avoid a spoofing attack where someone sets up an endpoint that looks just like yours, but with a rogue service running as some other user.

There are some details here <http://msdn.microsoft.com/en-us/library/ms733130.aspx>, although you will require some reading-around that article to understand what is actually going on; the config will ultimately look something like this:

```
<identity>  
  <servicePrincipalName value="CONTOSO\MyServiceUser">  
</identity>
```

MESSAGE SIZE

Controlling message size is another very important weapon in the armory against various kinds of attack, including denial of service. There's an excellent article on the subject in the MSDN documentation.

<http://msdn.microsoft.com/en-us/library/ms733135.aspx>

However, limiting messages to the default (64k) can be a little restrictive. It is a good idea to crank the message limits up a bit during load testing, and examine your average and maximum message sizes. This can give you an opportunity to set the size more appropriately for your app, and/or look at changing the granularity of some of your APIs if they are generating over-large messages.

PROXYING

The default web proxy is another standard config knob you may want to tweak.

```
useDefaultWebProxy="true"
```

Turning off proxying can be quite important in cross-domain scenarios where your local proxy can't actually see the server you're trying to reach.

IMPLEMENTATION

As we've mentioned several times already, service APIs are designed to perform one, atomic unit of work, and they will typically do so transactionally.

A typical service will instantiate a data context, perform some work (e.g. reading or writing information using that data context), and then submitting any changes. In the section on data access below, we will talk more about our choice of LINQ to SQL for the data access layer, and some of the techniques we use with that.

The data context automatically enrolls the work in a transaction.

```
using (IClinicalLocationManagementDataContext data =
    _dataContextFactory.CreateDataContext())
{
    DateTime serverTime = data.GetSystemDate();
    // Do some stuff
    // maybe data.SubmitChanges()
}
```

Using LINQ for your data access is extremely powerful, although it does take a certain “jump” in the developer mindset. To start with you ask questions like “how do I do a join” or “how do I do a correlated subquery”.

Eventually you realize that these are the wrong questions, and that you should be thinking in terms of your object model.

For example: although there is a “join” operator in the C# syntax, you more commonly just use the “.” operator e.g. *Patient.Names* “joins” patients to names.

Second, we make extensive use of projection. Just as you probably wouldn't

```
SELECT * FROM MYTABLE
```

Nor should you always

```
from a in data.MyTable select a
```

This can lead to unnecessary amounts of data coming back from the SQL Server. Instead, create a projection with the data you need in it

```
from a in data.MyTable select
new { UniqueId = a.UniqueId, SomeData = a.SomeData }
```

Also, beware of *DataLoadOptions* and lazy loading. Although they can provide you with some simple means of gathering additional information, a quick look in SQL Profiler can reveal a multitude of sins.

Particularly if you have a large number of items that reference a small number of distinct children (e.g. a thousand images with four different coded image kinds between them), it can still be faster to sort out the distinct items you need in code, go and fetch them in one small hit, and match them back up via a dictionary, than it is to flatten them into each and every row (or worse).

That said, LINQ to SQL produces, on the whole, better SQL than the average developer-in-a-hurry. Although you need to keep an eye on it (through SQL profiler), unless you start getting crazy with the *LoadOptions*, it will do a good job.

TIME

```
using (IClinicalLocationManagementDataContext data =
    _dataContextFactory.CreateDataContext())
{
    DateTime serverTime = data.GetSystemDate();
    // Do some stuff
    // maybe data.SubmitChanges()
}
```

I've picked out a line in bold in the example above, as it plays an important part in our service implementations.

Everything that goes into the store is identified by server time, not client time.

Even with clients that are automatically-synced to a clock somewhere, you can drift by several seconds, which can play havoc with concurrency decisions and other relative timing issues.

It is also a bad idea to call the moral-equivalent of *getdate* repeatedly inside a method, as the slight creep this can cause can also be a problem.

So, we expose a *GetSystemDate* method out of our data access layer which snapshots the server time at the beginning of the service call, and then use that one *DateTime* throughout the rest of our implementation.

```
[Function(Name = "GetDate", IsComposable = true)]
public DateTime GetSystemDate()
{
    MethodInfo mi = MethodBase.GetCurrentMethod() as MethodInfo;
    return (DateTime)this.ExecuteMethodCall(
        this,
        mi,
        new object[] { }).ReturnValue;
}
```

TRANSACTIONS

If there are several data contexts in play, or if we otherwise need to propagate a transaction, then we use the *System.Transactions* support for a transaction scope

```
using (IAppointmentManagementDataContext data =
    _appointmentDataContextFactory.CreateDataContext())
using (IWorkflowManagementDataContext workflowData =
    _workflowDataContextFactory.CreateDataContext(data.Connection))
using (TransactionScope scope = new TransactionScope())
{
    // Do some work on several contexts
    // Don't forget to call scope.Complete()!
    scope.Complete();
}
```

Note that the data contexts are actually using the same connection (highlighted in red above), so this does not even kick off a distributed transaction.

Note that there is no support for nested transaction scopes in LINQ to SQL (or, indeed, *System.Transactions*) – they will all inherit the ambient transaction, so if one aborts during, say *SubmitChanges()*, then the overall transaction is aborted – you can't just retry that one part.

SUMMARY

Our service layer uses WCF to expose atomic, stateless services to the clients. The service contracts are built using the standard WCF techniques of attributed interfaces, and business-layer objects. Those business objects support validation and data binding through *IDataErrorInfo* and *INotifyPropertyChanged* respectively.

We tweak the operation contracts using an attribute that extends the default serialization behavior to maintain references in the object graph during serialization.

Our service contracts are designed to balance message size, transactionality and usability, and we need to consider these issues at the client, client proxy, channel, and service layers. Paging, chunking and streaming are all important techniques.

DATA ACCESS

INTRODUCTION

Data access is very much one of those ever-changing issues. Back in the bad old days, we had live client connections to the database, streaming cursors through record sets. I say bad – this was extremely effective if you had an OK network connection and one or two clients; there was very little overhead for the connection, and life was good.

As time has gone by, we've had to find solutions to problems of scalability, disconnected operation, more easily maintained abstractions, new security models... the list goes on.

In our V1 application, there was a well-normalized database backend, although it had grown some slightly crusty bits over the year. There were some performance bottlenecks related to that normalization, and we had concerns about the size of some individual database tables.

While we did want to make some localized improvements in V2, we wished to avoid wholesale changes to the schema.

DATABASE SCHEMA

The principle change for new tables added to the system was a move from a table-as-its-own-audit-trail schema to a table-and-log approach.

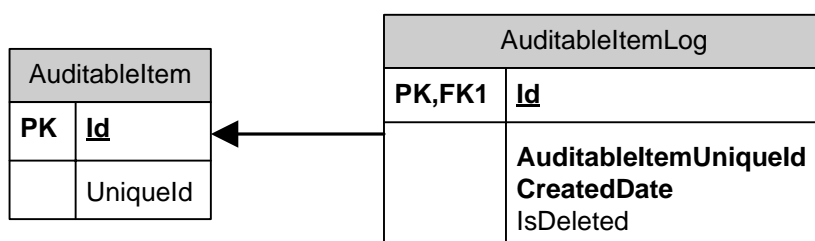
In V1, the tables look something like this:

AuditableItem	
PK	<u>Id</u>
	ValidFrom ValidTo

The *ValidFrom* and *ValidTo* dates determine the lifetime of the value. As the value is updated, additional *ValidFrom* and *ValidTo* rows are added to the table.

In terms of performance, this is fine where there are a few hundred thousand rows in the table, but it starts to creak as you get up towards the few million rows. It is also a pain in terms of retrieving “current” data – you constantly have to check the validity; it turns out, not surprisingly, that this is the most common case.

For V2, new entities are created like this:



The main table contains only the current, valid values. The log table contains a reference to the item in the main table, and maintains a creation date stamp. Both contain copies of the useful information in the row, whatever that may be.

When a row is added or updated in the main table, a new log row is also created. When a row is deleted from the main table, a log row is created with a snapshot of the original row's data, and the *IsDeleted* flag set.

Ultimately, this makes inserts slightly slower, and increases the total size of the database (as the values of an entity row are also cloned into its log), but decreases the cost of accessing the "current" version of data quite considerably.

Note that we also added *Uniqueld* columns to all of our entities, corresponding to the unique ids in our data model. This has required changes to quite a few of our V1 APIs which created *Uniquelds* corresponding to the entity *for the duration of the session*. We needed to fix many of these up so that they returned the "for all time" unique id that we now stored in the database.

Note that we still use an *int* as the primary key of our tables, and the *Uniqueld* as a secondary index. Although SQL Server 2005 now supports *uniqueidentifier* generation suitable for a clustered index, this would have been too great a change to our existing infrastructure.

Kimberley Tripp has a good article on this issue:

<http://www.sqlskills.com/blogs/kimberly/2005/06/06/EverincreasingClusteringKeyTheClusteredIndexDebateagain.aspx>

DATA ACCESS – TECHNOLOGY CHOICE

In V2 we considered a number of options, but focused on various object-based mapping frameworks.

Amongst the options under consideration were stored procedures and a simple, auto-generated wrapper to call them, to full blown ORMs like NHibernate (<http://www.hibernate.org/343.html>), LLBLGen Pro (<http://www.llblgen.com/defaultgeneric.aspx>) and Genome (<http://www.genom-e.com/>).

Then there are the "in the box" tools like LINQ to SQL and (nearly in the box, but not quite) LINQ to Entities.

The three main criteria we were looking for were ease of use, support for SQL Server 2005 & 2008, and support for LINQ.

NHibernate, LLBLGen Pro and Genome are clearly extremely powerful tools. They all support LINQ (although NHibernate is somewhat beta-ish), and are very good at providing a complex mapping transform from your ancient database schema to something more comprehensible. They can also support database platforms other than SQL Server.

Although we want to take advantage of some limited transforms on the original database schema (chiefly around naming conventions), we only need to support SQL2k5 & SQL2k8 (we have also dropped support of SQL Server 2000 in this version).

There is also the issue of whether we were looking to expose the model emerging from our database out through the service layer. The mapping technology supported by the more sophisticated ORMs allows you independently to maintain your database schema, and the data model to which it maps, opening up the possibility of exposing this generated model through the service contract and up to the client. Although this is possible with LINQ to SQL (by setting the *SerializationMode* to *Unidirectional*), the support is not sophisticated.

Personally, I am uncomfortable with this approach. During prototyping (when we experimented with NHibernate), we found that we were making savings, in terms of code mapping from our data access model to our business model, over that which we got from LINQ to SQL, and hand-rolling our business model / service contracts, but insufficient to take a dependency on another external codebase.

So, eventually, we came down on the side of LINQ to SQL. It is simple and well-integrated, although not without faults. The main problem is the ability to stub out the repository to talk to an in-memory data store of some kind for testing purposes.

TESTING A DATA CONTEXT

In order to provide data to unit test our services, we wish to stub out the data contexts to which they talk, to provide some controlled test data – in memory, and without hitting the database. If your (subset of) tests are taking more than a minute or so to run, then developers will gradually stop using them. Hitting the database is very slow, and, on the whole, we don't need to test that component; it is fairly likely that SQL server will work. We can also make the not unreasonable assumption that LINQ to SQL will work, too. It is just our usage of it that matters.

However, LINQ to SQL doesn't provide a convenient means of stubbing-out the provider (although there are some abstruse mechanisms out in the wild). This is a great shame.

To get around this problem, we define our own *IDataContext* interface which looks like this:

```
public interface IDataContext
{
    DateTime GetSystemDate();

    IDbConnection Connection { get; }

    IDbTransaction Transaction { get; set; }

    void SubmitChanges();
    void SafeSubmitChanges();
}
```

We then augment that interface with the features we need for our specific, generated data context:

```
public interface IAdvancedSearchDataContext :
    IDisposable, IDataContext
{
    ITestableTable<DemographicKind> DemographicKinds { get; }
    ITestableTable<Demographic> Demographics { get; }
    ITestableTable<PatientDemographic> PatientDemographics { get; }
    ITestableTable<Patient> Patients { get; }

    DataLoadOptions LoadOptions { get; set; }
    void SubmitChanges();
}
```

Note that a LINQ *EntitySet<>* in our generated data model maps to an *ITestableTable* in our interface.

```
public interface ITestableTable<TEntity>
    : IQueryable<TEntity>, ITable where TEntity : class
{ }
```

Notice that *ITestableTable* implements *IQueryable* and *ITable*, to support LINQ.

We then (explicitly) implement this interface in our code-behind partial class for the data context. For example:

```
ITestableTable<DemographicKind>
IAdvancedSearchDataContext.DemographicKinds
{
    get
    {
        return new TestableTable<DemographicKind>(this.DemographicKinds);
    }
}
```

Our *TestableTable* implementation delegates to the underlying LINQ entity sets.

For example:

```
public class TestableTable<TEntity> : ITestableTable<TEntity>
    where TEntity : class
{
    object _wrappedEntity;
    public TestableTable(object wrappedEntity)
    {
        _wrappedEntity = wrappedEntity;
    }

    #region IEnumerable<TEntity> Members

    public IEnumerator<TEntity> GetEnumerator()
    {
        return ((IEnumerable<TEntity>)_wrappedEntity).GetEnumerator();
    }

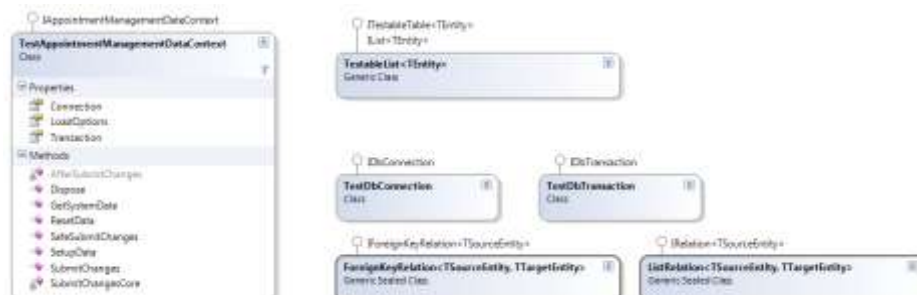
    #endregion

    // And all the rest...
}
```

So far so good – we’ve got a slim wrapper around our data context that effectively gives it an interface. So, down in our world of test, we can re-implement that interface, such that it doesn’t talk to the database at all.

Re-implementing the interface ranges from easy to “quite hard”.

We built ourselves some in-memory equivalents for the LINQ *EntitySet* and relation-management code, and coupled these with a list that deferred *Add* and *Remove* until a *SubmitChanges* method was called, to simulate the data store. We also needed a test connection and transaction.



TestableList is an interesting class. It is a reimplementation of the LINQ *EntitySet*<T> on an in-memory list.

The details of this implementation are well beyond the scope of this overview, but I hope this gives you an idea of the kind of code you need to build to mock around your service implementations.

We considered using the standard “replaying” mocking frameworks like MOQ (<http://code.google.com/p/moq/>) and NMock (<http://nmock.org/>), but, frankly, we didn’t invest enough time to fully explore their usefulness in this scenario.

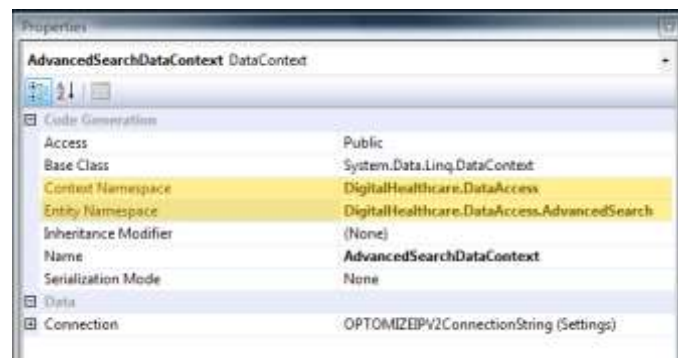
Instead, we decided to build a code generator, working from the same DBML files as generated the LINQ to SQL classes, to generate these test data contexts.

DESIGNING A DATA CONTEXT

There are quite a few tables in our database – about 200, in fact. This is way too many to build into a single data access model.

Instead, we partition the tables by “area of responsibility” (e.g. *PatientManagement*, *EncounterManagement*, *ClinicianManagement*), and create a separate data model for each.

Some tables will appear in more than one model. It is therefore important to set the Entity and Context namespaces to prevent collisions.



In all cases, we have used the mapping facility to sort out our database naming conventions, removing spurious underscores, camel casing and so forth. We’ve also changed the name of the primary key in the table from “Id” to “WhateverTheTableNameIsId” which makes it a little easier to read.

If a table doesn’t really belong to this model, but is referenced from it, we create an “identity only” stub table. This typically contains just the **Id (PK)** and the **Uniqueld**, and will ultimately be referenced in another data context.

While this slightly limits your ability to build LINQ-to-SQL queries across the boundaries, it makes the overall data access layer more maintainable and easier to read.

SUMMARY

Ultimately, the data access layer is fairly simple to build and maintain; the most difficult part is choosing your mapping technology of choice, determining how it is exposed through the services and designing it for testability.

We don’t expose the data access object model out through the services, but map it to our data contracts in the service layer itself.

Your choice of tooling will depend in part on how much of a transform you wish to make between your existing database and the object model, and how far up the stack you want to expose the results of the ORM.

We elected to use LINQ to SQL, and invested a considerable amount in abstracting out a version that could be used for testing.

You do not need to build every single entity into the one data model; it can be easier to understand and maintain a model representing only a subset of the entire database. Identifying “boundary” entities is critical to keeping it manageable.

CONCLUSIONS

The .NET framework V3.5 gives you a rich set of components to build applications that will live happily in a modern IT environment.

Because it builds on .NET 2.0, you do not need to throw away your existing investments; you can take advantage of WPF and WCF technologies side-by-side with your existing codebase, even co-habiting in existing Windows Forms applications.

However, the richer data binding model in WPF, and the ease with which you can build highly configurable web services using WCF encourages a “modularize and replace” strategy, rather than a fine-grained “add a WPF-based control here, a WCF service there”. You will get most benefit by building whole modules that take advantage of the new stack from top to bottom.

There is a considerable learning curve in WPF (it is a sea-change in the client tier), and in WCF (it is simple to get started, but there are so many buttons to push and dials to set that you can get buried in complexity when it comes to optimizing the configuration for a particular scenario), so taking on smaller projects as a test-bed is advisable.

It is also important to focus on delivering end user value. It is very difficult to get buy-in across your organization for a change in technology just for the sake of it. In this case, our V2 project was focused on delivering a compelling piece of end-user functionality, in a way that was quick to market, robust and with a richer user experience than would have been cost-effective with our V1 technologies.