

Re: OODesign – OPF, design pattern

Source:

<http://coding.derkeiler.com/Archive/Delphi/borland.public.delphi.non-technical/2007-08/msg03351.html>

- *From:* Aleksander Oven <aleksander.oven@xxxxxxxx>
 - *Date:* Wed, 29 Aug 2007 16:24:23 +0200
-

Peter Morris wrote:

Remember, don't confuse the concept with the implementation. A few of us have different implementations but generally they all do a similar thing, they let us develop applications in a more abstract way.

Maybe I am, I can't be sure. After all, it's the implementation that I see and have to deal with, usually. If it's bad, so is my opinion about the underlying concept.

What I had in mind were very specific design patterns, which (if you're using them) force you to write code in a round-about way. For example, when I hear someone tell me they're going to use the Visitor pattern to extend their class, I always cringe a little. Why can't they simply change the class in question? It's under their control, for goodness sake!

And then they go and change it – change it so that it accepts visitors, because it wasn't accepting them before, and that wasn't "elegant". To me, that's doing things just for the sake of things, and it's nothing more than a diversion from the boredom of conventional ways.

For the record, I'm not saying that's necessarily you, Peter (although your initial post made me wonder quite a bit :))! I'm merely observing my immediate reality.

I developed a Pocket PC app last year using the compact framework.

<snip>

It took a few months to write, especially as I also had to write my own OPF too, but it was definitely worth it! My boss moaned a bit "if this is a quick way to write apps why is it taking so long?" but once I had reached a point where I could stop "tweaking" the framework to handle scenarios that I hadn't previously thought of the development time suddenly increased dramatically.

Re: OODesign – OPF, design pattern

I can see a big problem with this. It may be fine for a single developer, or even two developers who think alike. But a team of developers can hardly be left waiting while one of them is tweaking the framework. It becomes a serious bottleneck!

My grudge with OO frameworks (not OO in general!) is that they are usually being developed as you go. And there's really no other way, because they are so broad (and potentially capable, I'm not denying that), they'd take forever if you'd try to implement them in their entirety before moving on to actual business logic of your product. And so there's always something critical that's left out – something left for the tweaking. :)

I'm all for using *some* of the OO principles to help build small and contained parts of the software, but not for everything.

It's a bad example, but I've seen a case of abstracting away the creation of the main form in a Delphi application a few years back. It took around a 100 lines of code to do it and was IMO a shining demonstration of what not to do. And yet, I couldn't help but be a little impressed. Almost like with a bad car pile-up – you just have to wonder how it happened. :)

I think it is definitely more work in Win32 than it is in .NET.

I can imagine, yes.

Additionally, at first I find it quite slow, but once you know how your code looks you can use something like ModelMaker and then write a code-gen hook to emit additional code based on diagrams etc. I have a code-gen plugin that generates all of the framework members for my properties and associations.

Oooh, another pet peeve! Code generators.
<sarcasm>Tools that help me solve a problem I probably wouldn't even have if I'd just stay away from complex frameworks.</sarcasm>

Seriosuly, though... They /can/ be beneficial. But they can also inflict hidden maintenance costs if there's too much code. And some frameworks out have code generators there are spewing out enormous amounts of auto-code.

In reality, I don't see much difference in modifying a class diagram and modifying a class manually. Properties and methods are hardly too complex to write. A properly configured template expansion in modern IDEs can save just as much time as a separate dialog where you need to enter 80% of the same strings anyway.

Re: OODesign – OPF, design pattern

designing the
whole application around such decoupled layers? That's just insane!

Why? The most simple way I think is to do this

<GUI>
<Object types the GUI uses>
<Application API>
<Database>

Sure, if you boil it down to this level, it's almost the same diagram for both OO design and the conventional design. But we both know that once you zoom in even just a little, OO design reveals more layers.

But it's not really about the number of layers. I'd be wrong to leave it at that. The emphasis of my concerns is really in the way they are coupled (or decoupled, to be precise). OO design dictates that layers be as much isolated and abstracted from one another as possible. This ensures that they are flexible and that their internal implementation can easily be swapped for another.

While an admirable goal, I find it extremely rare that it actually saves any time, it just deflects the focus of changes to another place. For example, say you're using the .NET grid control, but you need a more feature-rich control. So you decide to switch to DX XtraGrid. With a conventional design, you need to rewrite all the event handlers and all the code that's calling the grid control in any way. With OO design, you need to change the adapter class in your View that's hooked up to those same event handlers and, if you're lucky, only the implementation of the adapter's public interface. But it turns out you're not that lucky. You need a new interface on your adapter, because in your initial design, you didn't anticipate that you'd be switching to so radically different grid, and as such the adapter's original interface isn't compatible anymore.

You may call my example a stretch, but I've seen plenty of this. Having OO framework that's capable of adapting to the wildest imaginable range of changes in theory, and in the end still having to change just as much code (but usually more, because every real-life framework is leaking abstractions to outer layers, which in turn also need to be adjusted).

A few years back, I tried to implement a small application this way. What constantly kept getting in my way was the fact, that UI controls and the RTL and VCL itselfs simply aren't built to support MVP/MVC-style of development.

Re: OODesign – OPF, design pattern

Your middle–man could take the form of a TDataSet.

My problems weren't limited to a databinding level. They were everywhere, e.g. in handling drag–n'–drop, responding to focus changes... You know, all the little things that all 3rd–party controls like to handle internally.

I ended up having a whole lot of abstraction leakage between the layers and was finding myself constantly having to resort to hacks to minimize those leaks.

I never use interfaces to reference count. I've never seen an approach I really like.

Um, I think you've misunderstood what I was saying. Either that, or I've just misunderstood what you're saying. :)

I wasn't talking about memory leaks. By "leaking abstractions" I meant this: <http://www.joelonsoftware.com/articles/LeakyAbstractions.html>

It's a real problem when layers, which were supposed to be isolated from one another, share knowledge about their internals. This automatically means that any non–trivial change in one of them will require some kind of change in the other. This means an annoying distraction at best and a maintenance nightmare at worst.

Or go with the TDataSet wrapper ofcourse :–)

I'm not sure I understand what you're getting at, but I'll repeat: it's not just about the databinding layer. It's the hidden internals of every rich UI control out there. You often need to respond to interim changes that are happening inside the control, while user is interacting with it, but you can't get to appropriate status indicators to actually do it. And so you are forced to hack.

I've never written wrappers. I either databding (ECO) or my composite GUI control knows how to display the request object and also how to set the response values (my compact framework app).

Ah, so you *do* write them! You're just calling them "composite UI controls". :)

Re: OODesign – OPF, design pattern

I guess it's not unusual to see OO-experts writing their own TEdit and TButton implementations.

I've never done that.

Yeah, I knew I was over-simplifying. :)
But I have seen it done. Poorly, I might add.

you sometimes double and even triple
the amount of code that needs to be written in order to achieve even
the simplest of tasks.

I imagine it can be a lot of work, yes. The benefit I suppose is that you only ever have to do it once. It's not something I've ever had to do though.

I almost read this as "It's not something I've ever /been able/ to do".
:)

'Cause that's the way I see custom frameworks. They're usually only usable for one application in the end and have to be adapted for the next one. Again, I'm not talking about commercial frameworks that expert people spent years perfecting, but all the custom ones that are being written in-house.

I haven't yet seen an OO-based* application, that wasn't at least twice as hefty than its plain counterpart.

My PPC one isn't.

<snip>

<impressedMode>It certainly looks like a nicely distributed framework.</impressedMode> :)

Sure, if you go .NET-style and say that framework doesn't count as weight, it really boils down to 3 small assemblies. But then again, it's still you who has to maintain those other assemblies ("the background", as Joanna would say). I still count this as being at project's expense, unless you really have quite a few projects that are using your framework exactly as it is. Then I agree it's definitely worth it.

Usually though, I see frameworks being developed with /intention/ to be used anywhere, when in reality, they're only ever used once or maybe

Re: OODesign – OPF, design pattern

twice, provided some modifications are made first.

* This is actually one of my pet peeves. OO used to mean "Object Oriented", but nowadays it's more a synonym for "Object Persistence Framework Oriented".

Nah. I do plenty of OOP that is nothing to do with OPF.

<snip>

OPF just utilises OOP. If you write OOP then you are writing OOP :-)

Yes, I really meant to say "OO design" here, as I've already explained to Joanna.

Maybe my design is just very simple then, because whenever someone asks me for a new feature I just grin and say "Okay". For example

<snip>

I'm looking forward to it, and its in the application layer not the business objects layer. I wont have a "ThreeInOneOrder" class or anything, just a new task in the app layer that I will execute instead of the existing ones.

It very much depends on the kind of a feature. :)

In my experience, the features that develop-as-you-go OO-frameworks have most problems with, are the variations on the same theme. If they are vastly distinct, it's less of a hassle, because you can just extend the framework. Adapting existing parts of it, OTOH, is quite something else, because it breaks the initial design. :)

You become locked in by your own clever design!

Or locked in by an over complicated design maybe?

Yes, "clever" was supposed to be read in a sarcastic tone. :)

and it's all interfaces, it's really hard to get additional information about the actual object that's misbehaving. Before you know it, you need to write additional code to assist you in debugging. Yeah, more code – that will help fight complexity...

Re: OODesign – OPF, design pattern

I don't use interfaces that much. I do use them, but I do use them when I feel I need them. You could always have them all implement an IAsObject interface with the following

```
function GetAsObject: TObject;
```

Then you can grab the object itself for debugging purposes.

That's not nearly enough! In any moderately complex OO framework, you'll have a range of common ancestor classes that implement the core interfaces and patterns. TObserver, TAspectObserver, TActor, etc. Then, other classes will either descend directly from those, or indirectly incapsulate them as to implement a sort of multiple inheritance. For example, there's often need to have a class that behaves as Observer, Subject and a CommandLink in a chain of command, all combined. GetAsObject() does you very little in such case, as you still need to know the exact class to cast it to, before you can get to the internals. An sometimes it's not even the class itself that you want, but its container! And in a long call stack it can be very tedious to analyze every line in this way to determine exactly which objects have been talking to each other. So you either write redundant code upfront and have more classes that all implement different combinations of the same interfaces, or you write debug-assisting code in the aftermath. Pick your poison. :)

And then there are the side effects – unforeseen interactions between parts of the program that weren't supposed to affect each other. It turns out infinite notification loops are all too common when complexity grows. And there's no definite counter-measures to fight them, except for iron discipline during development, and – you guessed it – more code! Checks on every critical entry-point to make sure we don't end up in a notification cycle of doom.

I really don't know what you are talking about here, I've never experienced such a thing.

:)

Maybe the problem is that you are trying to throw in as many design patterns as you possibly can? I use very few design patterns, and with multi-cast events in .NET I use one less :-)

I don't think that's the case. The example I gave was a side-effect of just the Observer pattern. And I'm sure it wouldn't have happened had I contained its use to smaller parts of the program. The mistake, IMO, was implementing it on the MVC level and using the MVC design on its own.

Re: OODesign – OPF, design pattern

I can't help but feel that this extreme OO is just one of the many variations on the same theme in software industry – emperor's (relatively) new clothes.

It's only that case if it is pointless. For you it may be pointless, but for others it isn't :-)

Sure, to each their own.

My problem is, I've seen no convincing real-world examples where it actually made sense to do things so radically differently from the conventional ways. It all seems to be just focus-shifting in the end – do you make all your changes here, or do you make them there (and there... oh, yes, and over there, too :))?

I wonder how much time it would take me to if I'd have written it using OO principles in the first place? :D

At least a year, you'd have to write your "blog writing framework" first :-)

Lol! Excellent point. :)

—

Regards,
Aleksander Oven

.