Understanding Business Objects, Part 1

A well-designed set of business objects forms the engine for your application, but learning to create and use business objects has been a struggle for this author.

Tamar E. Granor, Ph.D.

I've been hearing about business objects since some time in the mid-1990's. Not long after VFP added object-orientation, people started recommending that business logic be encapsulated into a set of separate objects. Intellectually, I understood the idea, but the examples I saw never really seemed to deliver on the promise. The standard example involved a customer object with the customer data entry form calling on that business object to do things like calculate sales tax. While I could see how to build that kind of object, it didn't seem all that important.

Several years ago, I started working on a highly graphical application, where the forms don't look like standard data entry forms, the business objects needed to represent real, physical objects, and where there wasn't a simple mapping between business objects and forms. As I worked on this application, especially as requirements changed, the power of business objects really became apparent to me.

Over the next few issues, I'm going to take a fresh look at business objects from that perspective. I'll use a highly graphical application, a Sudoku game, to demonstrate.

What are business objects?

The first FoxPro book I encountered that talked about business objects was "The Visual FoxPro 3 Codebook" by Y. Alan Griver. Interestingly, though it provided business object classes, nowhere did it actually define the term "business object." That book came out in 1995, but we're not doing much better at actually defining what we mean by "business object" today. Here are a few definitions of business objects found on the web.

From Wikipedia:
Business objects are objects in an object-oriented computer program that represent the entities in the business domain that the program is designed to support.

From PCMag.Com's online encyclopedia:
A broad category of business processes that are modeled as objects. A business object can be as large as an entire order processing system or a small process within an information system.

From ObjectMatter.Com:
An object that is modeled after a business concept, such as a person, place, event, or process. Business objects represent real world things such as employees, products, invoices, or payments.

The first two definitions are more circular than helpful. The third, though, offers some idea of what we mean when we describe something as a "business object."

Let's expand on that by looking at what we expect a business object to do.

First, we want it to contain data that describes the real world thing represented. For example, a product business object might have properties for the product code, the product description, the unit price of the product, and so forth. To database developers, this sounds like a record, not an object. Where it varies, though, is that a business object's properties need not be scalar. That is, it can contain arrays and collections, as well as references to other objects.

Second, it should operate on that data in ways that model real world activities. For example, an employee business object might include a method to compute the number of days of service of that employee.

So a business object differs from a record representing the same object in two ways, the ability to contain non-scalar data and the inclusion of methods that operate on the business object.

How does a business object differ from any other object? It doesn't really, except that it normally represents some real-world entity, not just a programmatic abstraction.

Why business objects?

The standard explanation given for why you should use business objects is that you should "separate interface from implementation." That is, the logic behind the application (the implementation or "engine") should be separate from the code used for display (the interface). But that's really just another way of saying that you should use business objects. It doesn't explain why.

What are some real-world, practical reasons for using business objects?

The most important is that it lets you put code in one place. That method to compute the number of days of service for an employee might be needed in several places in your application (the employee maintenance form, a seniority report, and so forth). With an employee business object, you put it there, and you always know where to find it. Without business objects, you end up writing the code for the employee maintenance form, then writing it again (or cloning it) for the seniority report, and again for the next use. (Yes, you might realize that you're duplicating code and write a function instead. I'd argue that, in some sense, such a function is part of a business object, even though it's not technically inside an object.)

Another related reason to use business objects is that it encourages you to put all the business logic in one place. This is the flip side of having a particular piece of code only once. When business rules are implemented in the user interface, logic tends to be scattered all over the place. Then, when you need to figure out how something works, or why it works as it does, you need to hunt for it. When business logic is restricted to business objects, all the code that relates to a particular process is in one place (or, more likely, a few places, if you have several cooperating objects).

The practical reason I most often hear for using business objects is that it lets you vary the user interface or the back-end database without having to rewrite code. While the forms in a desktop application aren't usually replaced wholesale once the application ships, more and more applications do need to run both on the desktop and on the web, or are migrating from the desktop to the web. Similarly, there may be situations where it makes sense to change the database used for an application. In particular, if security or reliability considerations change, or the amount of data involved grows more than expected, moving from VFP native data to a SQL back-end may be called for. In both of these cases, having business logic in business objects makes the transition much easier than if business logic is tied tightly to the user interface.

The problem with the usual examples

Though I understood the reasons for using business objects, I never really felt comfortable with them. Using them often felt like it just added a layer of indirection to developing applications.

Part of the problem was the way they were presented to the VFP community. Our first exposure to business objects came from the Codebook framework and the other frameworks that built on that one. Perhaps because it was the first attempt at using business objects with VFP, the approach is somewhat clumsy and too tightly integrated with the user interface.

Codebook's "base" business object class is a subclass of the Container class, a visual class. In fact, business objects are visible in Codebook applications and contain the controls used to work with a given business object. I think it was this aspect of Codebook's approach that made it hard for me to see why I'd bother.

Codebook's base business class also wraps up a lot of database functionality, providing methods to move through a set of records represented by business objects, as well methods for standard database functionality, like creating, saving and deleting records. It's designed to make it possible to work with local VFP data and remote data interchangeably. While this is, of course, a good thing, in some ways, it also makes the class seem like a poor cousin to VFP's native database functionality. That's especially true if you're working exclusively with native VFP data.

Several other VFP frameworks (including Visual FoxExpress, known as VFE) are based on the Codebook model, and perpetuate the idea of a container for a business object (though VFE does not expect you to put any controls in the business object). The Visual MaxFrame Pro framework, developed independently (though likely influenced by Codebook) takes the same approach.

Eventually, though, the VFP community realized that tying business objects to user interface was a bad idea, and later frameworks separated the two, subclassing the "base" business class from non-visual classes. For example, the Mere Mortals framework, though it's a descendant of Codebook, uses a non-visual class for its base business class. The focus on managing data remained, of course.

A major idea in all these frameworks, though, is that you create business objects that contain the business logic of your application and then drop these classes onto forms. In Codebook, a form has a single business object, which contains not only the business logic, but also the necessary controls. In VFE, you add business objects to presentation objects; the presentation objects contain the actual controls and are placed on forms. Mere Mortals can handle multiple business objects on a form, but still thinks of them primarily as something related to the form as a whole, not to specific contents of the form.

How I got it

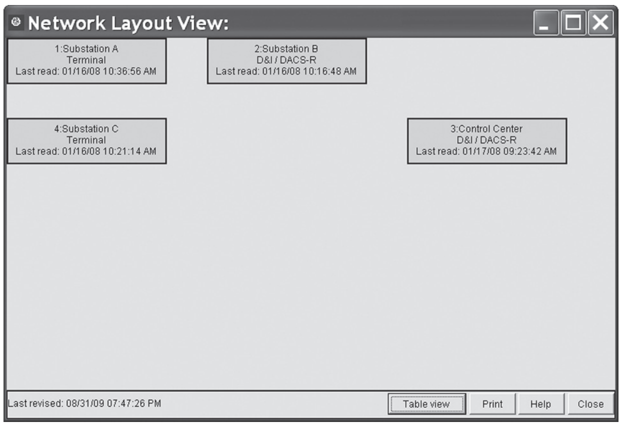
The examples that first helped business object concepts gel for me didn't actually refer to "business objects." Around VFP 6, the "Xbase tools" that come with VFP (tools written in VFP; they're now part of VFPX) started to use a model of separate interface and implementation objects. For example, the Coverage Profiler includes a cov_engine class (the implementation object) and a cov_frame class (the user interface). The idea was that you could subclass the two separately, so you could change either the behavior or the way it looked to the user or both independently.

This idea made enough sense to me that I soon applied it to someone else's work. Doug Hennig published an article in the January, 2000 *FoxTalk* showing how to make your applications remember things like where a window was last positioned and what record it was looking at. Doug set it up as a single class hierarchy, with subclasses to handle variations like where to store the data to be remembered and what data was to be stored. When I decided to use the technique, I found that there were two things I wanted to vary independently, where to store the data and what to store. So I refactored Doug's class hierarchy to create two separate hierarchies, one for the "persistence engine" that keeps track of what data to remember and handles types conversions and so forth, and a separate one to do the actual storage and retrieval, that is, to be the interface (though not a user interface in this case) to a storage device. (I wrote a little bit about these classes in the November, 2002 issue of *FoxPro Advisor*.)

Even after that exercise, though, the kind of business objects I found in all the frameworks still felt clumsy. I began work on my own framework, and included business objects built from the CursorAdapter base class. Like the other frameworks, my business classes incorporate methods for moving through a data set and handling basic record operations like new, save and delete. But using those classes still felt like programming with gloves on.

Then, a really unusual application landed on my desk, a network management system (NMS) used to monitor and manage multiplexers in utility substations. The application, originally written in FoxPro 2.6 and ported to VFP 6, was well-designed but showing its age. The client wanted to maintain its functionality, but add a far more graphical front-end while moving to VFP 9.

The application addresses a network of nodes, with each node representing a substation. Within each node, there can be one or more shelves, which in turn contain circuit boards. Each circuit board has a bunch of settings that can be read or written through the application. The goal for the updated application was to show the networks and nodes graphically, while providing an easy way to edit the settings for a given circuit board. The new form representing the entire network (Network View, shown in Figure 1) shows each node in the network and allows the user to rearrange them to make sense, as well as to add connections between them (though the figure doesn't include any). The form representing a node (Node View, shown in Figure 2) looks much like the actual hardware at the node, showing each shelf and each circuit board within. Only the form for the settings of a particular board (Card View, shown in Figure 3) looks anything like a conventional data entry form, but its contents and behavior are driven by meta-data.


Network Layout View:

1:Substation A
Terminal
Last read: 01/16/08 10:36:56 AM

2:Substation B
D&I / DACS-R
Last read: 01/16/08 10:16:48 AM

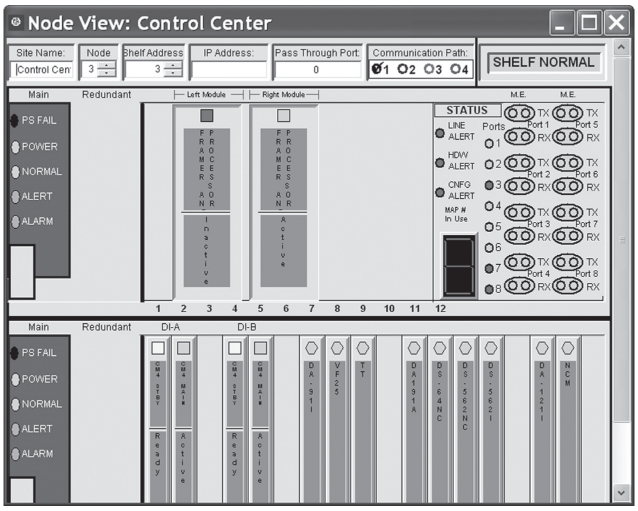
4:Substation C
Terminal
Last read: 01/16/08 10:21:14 AM

3:Control Center
D&I / DACS-R
Last read: 01/17/08 09:23:42 AM

Last revised: 08/31/09 07:47:26 PM

Figure 1. In this highly graphical application, Network View shows a summary for each node (substation). Nodes can be dragged to position them according to their actual relative locations. Color indicates status of the node. Lines can be added between nodes to represent actual connections between them.

More importantly, each of the forms needs access to information at multiple levels. Network View addresses the network as a whole and the individual nodes. Node View talks to a node, its shelves, and the boards they contain. Card View talks to an individual board, but also needs some shelf and node information.


Node View: Control Center

Site Name: Control Cen
Node: 3
Shelf Address: 3
IP Address:
Pass Through Port: 0
Communication Path: 1 2 3 4
SHELF NORMAL

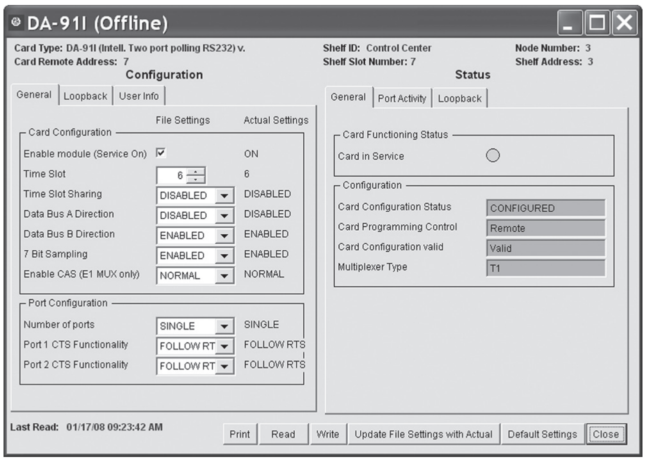
Figure 2. Node View shows all the shelves and boards of one node. In this version, boards (also known as "cards") can be dragged from one slot to another.

When I began to work on the new forms, I started with the graphical aspects, using VFP's containers, shapes, lines and labels to construct the objects I'd need. Once I'd done that, I needed to find a way to connect the actual data to its graphical representations.

The data is stored in a reasonably normalized set of tables. Four tables store the bulk of the information. Network contains a single record with network-level data. Node contains one record for each node in the network. Slot has one record for each board and Values has a record for each individual setting.

This means that Network View represents a single record from Network and multiple records from Node; Node View contains data from one record in Node, from multiple records in Slot, and has some data from Values; and Card View represents multiple Values records with some data drawn from Slot and Node. It rapidly became clear that I'd need some kind of data structure to enable me to pull all the data for each form together.

Enter business objects. I created a set of classes to hold the data and gave them methods to move the data in and out of the tables. So, when the user picked a network to work with, data from the tables was immediately moved into this hierarchy of objects. The forms talk only to the objects, never to the tables.



Card Type: DA-91I (Intell. Two port polling RS232) v.
Card Remote Address: 7
Shelf ID: Control Center
Shelf Slot Number: 7
Node Number: 3
Shelf Address: 3

Configuration

	File Settings	Actual Settings
Card Configuration		
Enable module (Service On)	✓	ON
Time Slot	6	6
Time Slot Sharing	DISABLED	DISABLED
Data Bus A Direction	DISABLED	DISABLED
Data Bus B Direction	ENABLED	ENABLED
7 Bit Sampling	ENABLED	ENABLED
Enable CAS (E1 MUX only)	NORMAL	NORMAL
Port Configuration		
Number of ports	SINGLE	SINGLE
Port 1 CTS Functionality	FOLLOW RT	FOLLOW RTS
Port 2 CTS Functionality	FOLLOW RT	FOLLOW RTS

Status

Card Functioning Status
Card in Service ○

Configuration

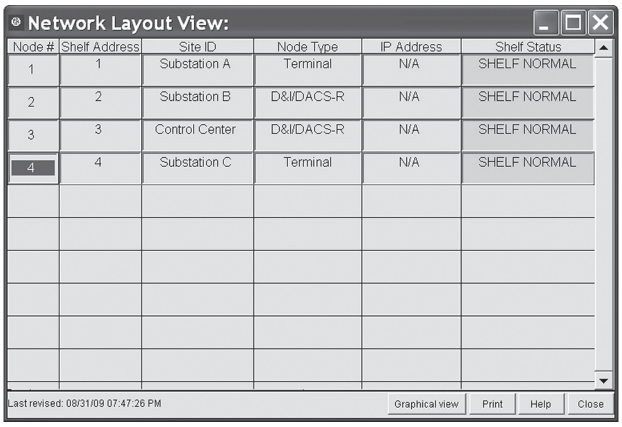
Card Configuration Status	CONFIGURED
Card Programming Control	Remote
Card Configuration valid	Valid
Multiplexer Type	T1

Last Read: 01/17/08 09:23:42 AM

Figure 3. Card View shows the settings for an individual circuit board; the pages, boxes within the pages and items within the boxes are all driven by meta-data.

As I got deeper into implementing all the required functionality, I found myself adding more and more methods to these classes to answer questions about the state of things or to allow the user to change the contents of the network. Once the client started working with the modified application, and requirements were changed or refined, these classes continued to change and grow. More importantly, implementing most of the changes turned out to be fairly simple.

One major change was to provide a second, much less graphical, format for Network View (shown in Figure 4), intended primarily for use with large networks. Because the data in Network View was coming from business objects, and form methods were used to communicate with the business objects, creating this version was straightforward.

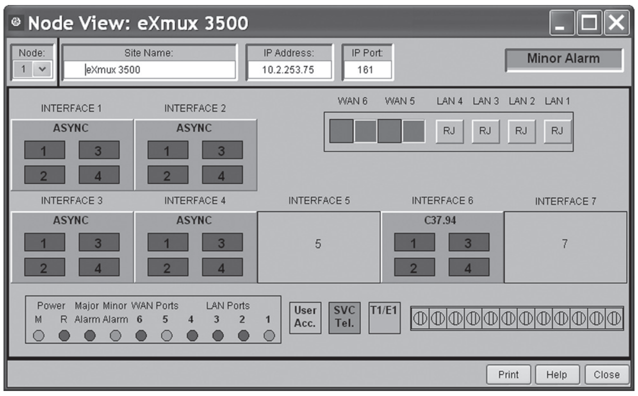


Node #	Shelf Address	Site ID	Node Type	IP Address	Shelf Status
1	1	Substation A	Terminal	N/A	SHELF NORMAL
2	2	Substation B	D&I/DACS-R	N/A	SHELF NORMAL
3	3	Control Center	D&I/DACS-R	N/A	SHELF NORMAL
4	4	Substation C	Terminal	N/A	SHELF NORMAL

Last revised: 08/31/09 07:47:26 PM

Figure 4. Creating this alternate version of Network View was a breeze, because all the data was drawn from business objects. Only the graphical portions had to change.

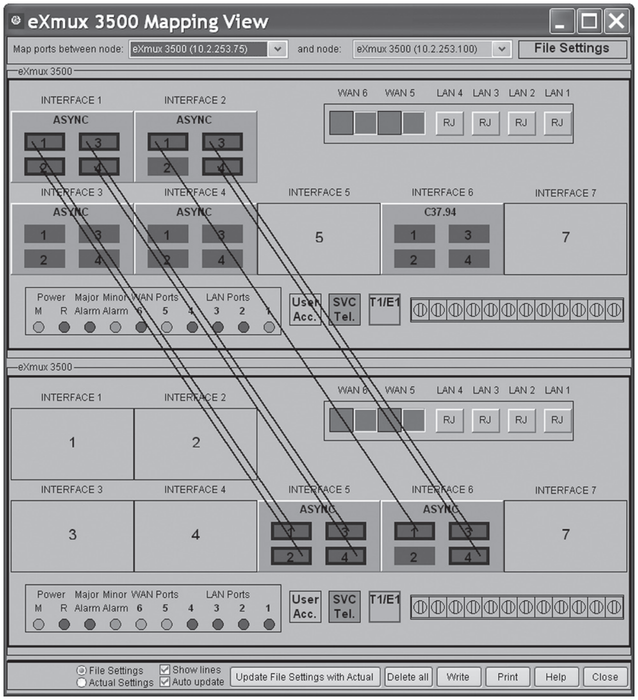
As this project was being completed, the client was preparing to introduce a new line of multiplexers that would need its own Network Management System. While many requirements for the new system were the same as the old, there were also a number of significant changes. Where the old multiplexers could handle 18 circuit boards each, the new ones have only 7 slots and a single board can stretch across 2 slots. In the old system, the application handles communication with the actual hardware. The new application would talk to another software layer that would handle communication with the hardware. The old multiplexers work with several different kinds of shelves, with a single node able to house several shelves. Using the new multiplexers, a node can contain only a single shelf and all shelves have the same architecture. However, the client wanted to preserve the possibility of multiple shelves in a node for the future. Clearly, Node View would be quite different in this version; Figure 5 shows the updated Node View. The client also wanted some changes in Network View and Card View, plus a new form, similar to Node View, to show the relationships between the nodes (see Figure 6).



Node: 1
Site Name: eXmux 3500
IP Address: 10.2.253.75
IP Port: 161

Figure 5. In the new version of NMS, Node View has changed quite a bit, but the object model underneath is nearly the same as in the older version.

When we started adapting the updated NMS to create an NMS for the new multiplexers, the power of business objects became even more apparent. Using our existing code as a base, we were able to get a prototype of the new version up and running in just a few weeks, so the client could demonstrate it at a trade show. Because this was a brand new product for them and it was still under development as we were working, requirements changed quite a bit over time; again, the decision to use business objects meant that most changes were able to be handled quickly and easily. (In fact, as I was writing the white paper on which this article is based, I was working on major changes in one area of the object model. Even though this area needed two significant changes that were not anticipated in the original design, much of the code was able to be used unchanged, and changes to many other methods were small. As I've found throughout this project, the larger effort was more often figuring out what to change and how than writing the actual code.)



Map ports between node: eXmux 3500 (10.2.253.75) and node: eXmux 3500 (10.2.253.100)

Figure 6. Mapping View is new in the most recent version of NMS. It shows the relationships between nodes. Thanks to the business objects underneath, it shares a lot of code with Node View. Both forms are subclassed from the same parent class.

Now, more than three years after taking over the initial project, this set of objects is second nature to me and it's hard to imagine that there could have been any other way to implement this system.

Designing and Implementing Business Objects

My experience with NMS taught me to think of business objects as the engine for my application, allowing forms to serve only as a conduit for the user. The ideas that felt so clumsy when looking at a standard database application become much clearer in a graphical setting where a particular object could appear in many different forms.

In my next few articles, I'll explore another (simpler) application, and show how to decide what business objects to create, what to put into them and how to link them to the user interface. Finally, I'll look at making changes to an application and how business objects smooth the way.

Author Profile

Tamar E. Granor, Ph.D. is the owner of Tomorrow's Solutions, LLC. She has developed and enhanced numerous Visual FoxPro applications for businesses and other organizations. She currently focuses on working with other developers through consulting and subcontracting. Tamar is author or co-author of ten books including the award winning Hacker's Guide to Visual FoxPro, Microsoft Office Automation with VisualFoxPro and Taming Visual FoxPro's SQL . Her latest collaboration is Making Sense of Sedna and SP2, coming out this year. Her books are available from Hentzenwerke Publishing (www.hentzenwerke.com). Tamar is a Microsoft Support Most Valuable Professional. In 2007, Tamar received the Visual FoxPro Community Lifetime Achievement Award. You can reach her at tamar@thegranors.com or through www.tomorrowssolutionsllc.com.