Getting Your Head Around Business Objects

Tamar E. Granor
Tomorrow's Solutions, LLC
8201 Cedar Road
Elkins Park, PA 19027
Voice: 215-635-1958
Email: tamar@tomorrowssolutionsllc.com

For many years, we've heard that business objects are important, but most of the examples are tied too tightly to the user interface to really make the point. In this session, we use a highly graphical example to show how business objects can really improve your applications. We see how a well-designed set of business objects makes changing an application's behavior easier and keeps your application's functionality separate from its user interface.

I've been hearing about business objects since some time in the mid-90'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 and 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.

This paper takes a fresh look at business objects from that perspective. It uses 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:

- 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.

- 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 a very 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) 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 playacting.

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 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.

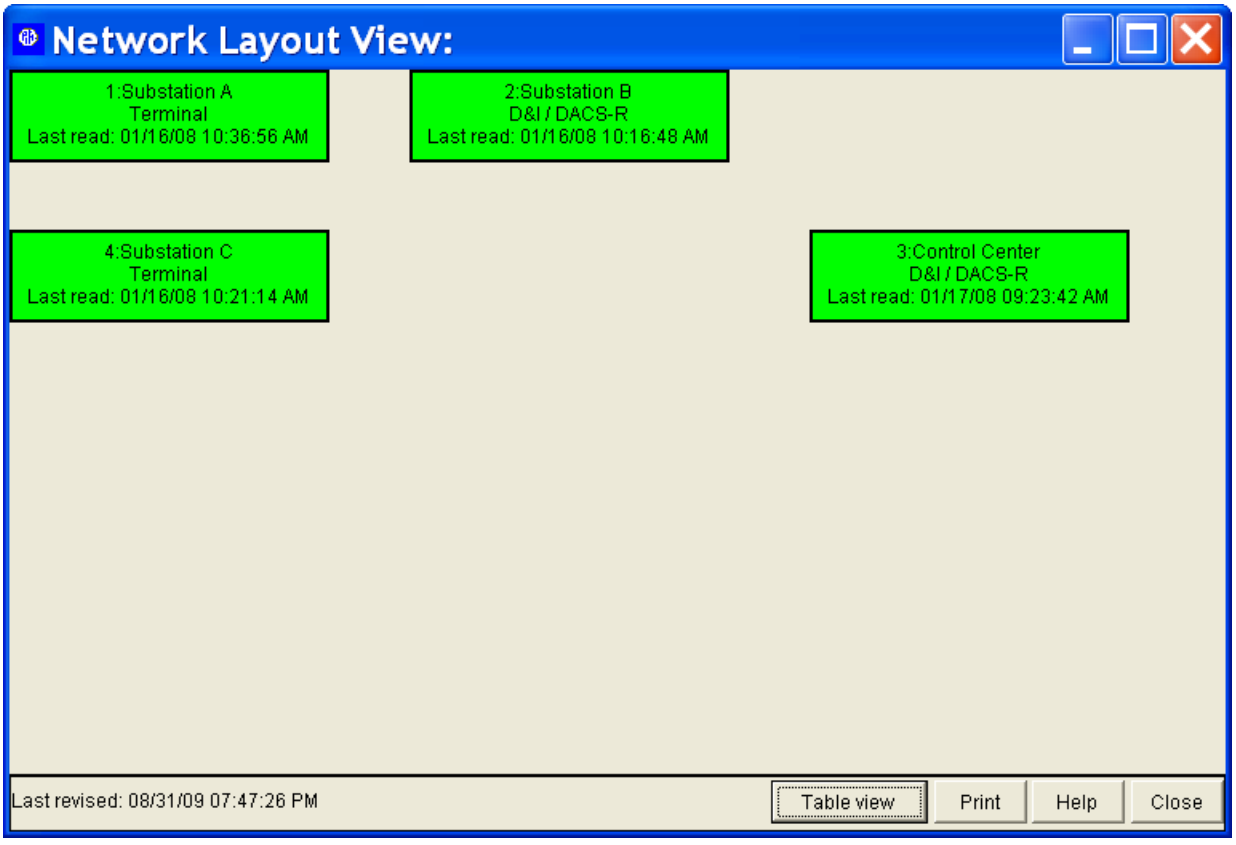

Figure 1. In this highly graphical application, Network View shows a summary for each node (substation). Nodes can be dragged to position them according to their actual relative locations. Color indicates status of the node. Lines can be added between nodes to represent actual connections between them.

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.

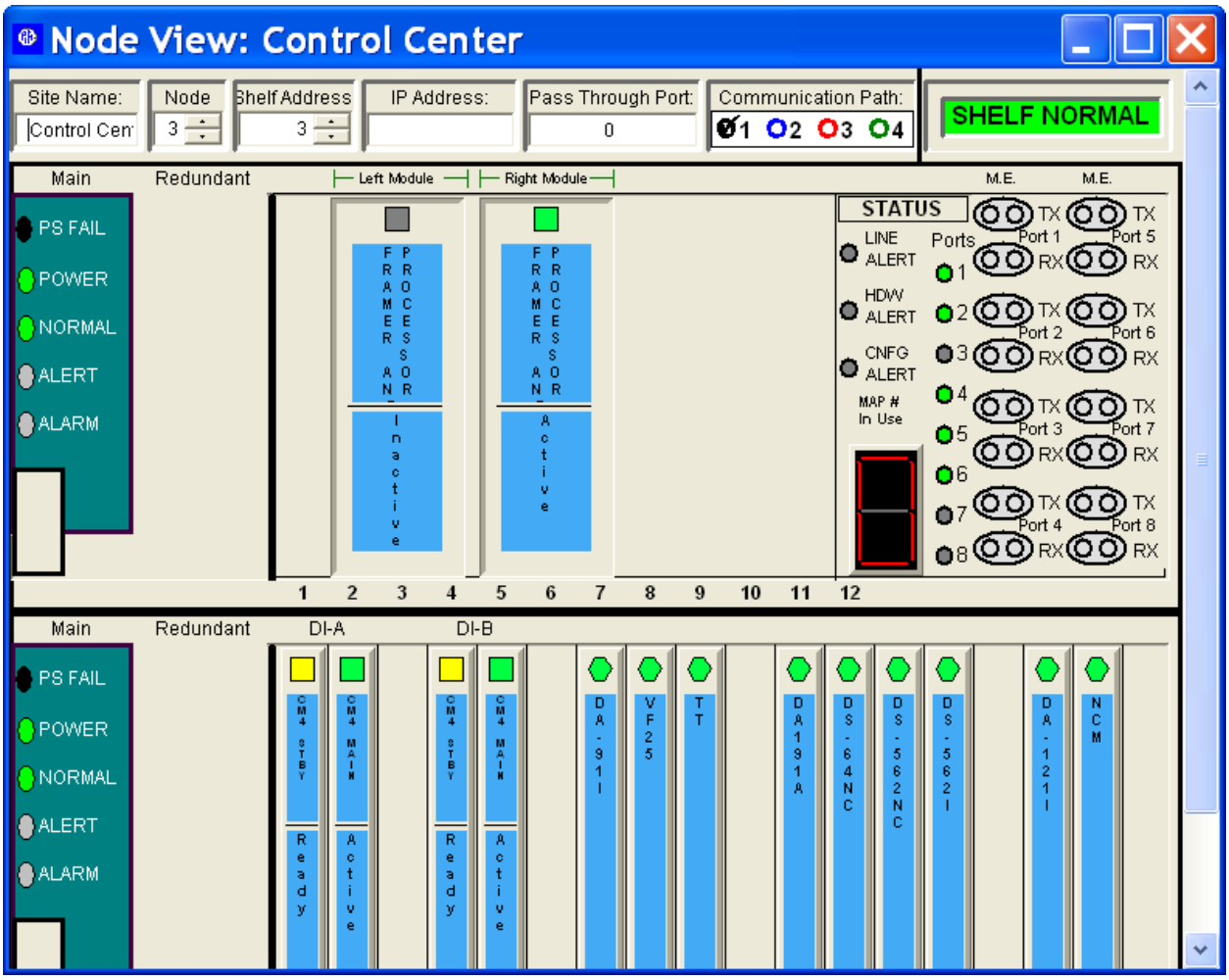

Figure 2. Node View shows all the shelves and boards of one node. In this version, boards (also known as "cards") can be dragged from one slot to another.

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 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. 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.

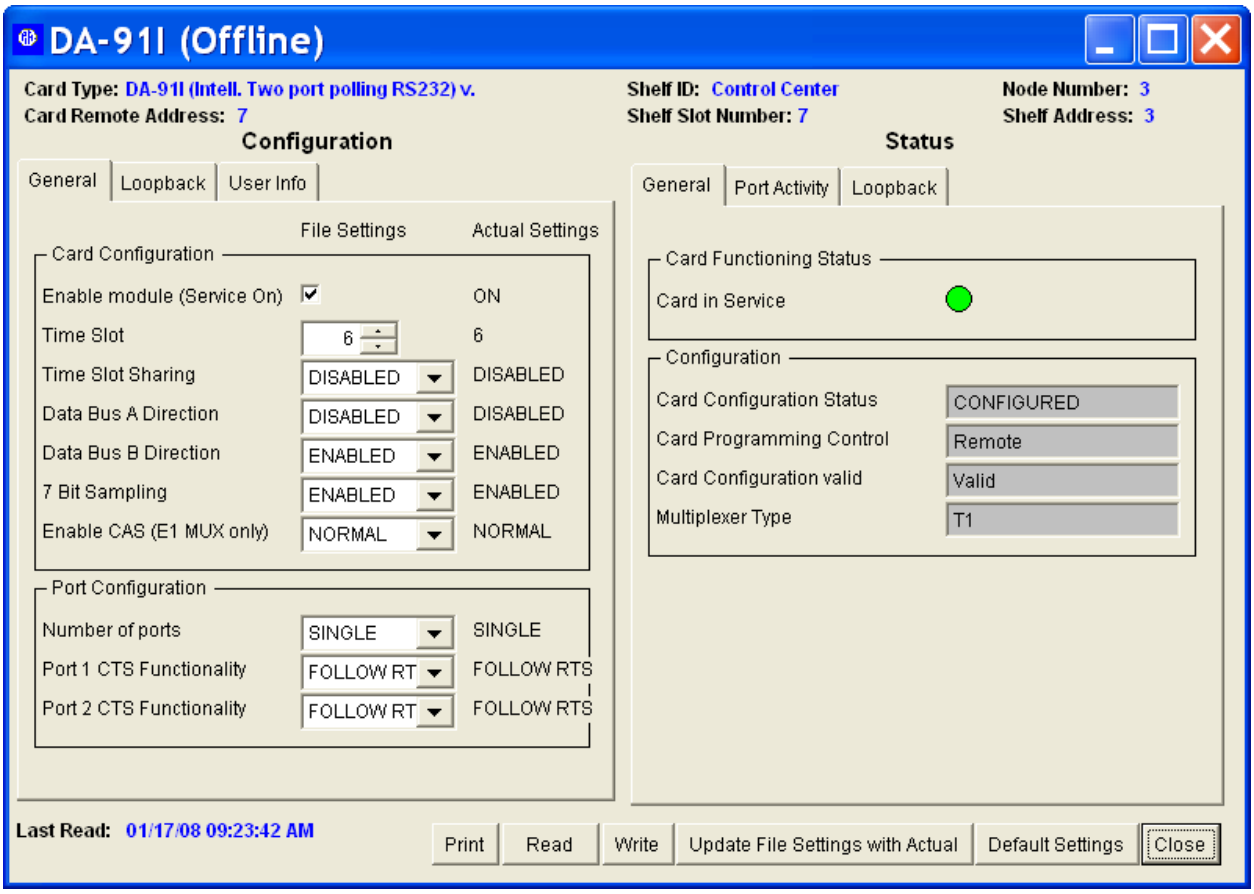

Figure 3. Card View shows the settings for an individual circuit board; the group of settings shown is driven by meta-data.

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.

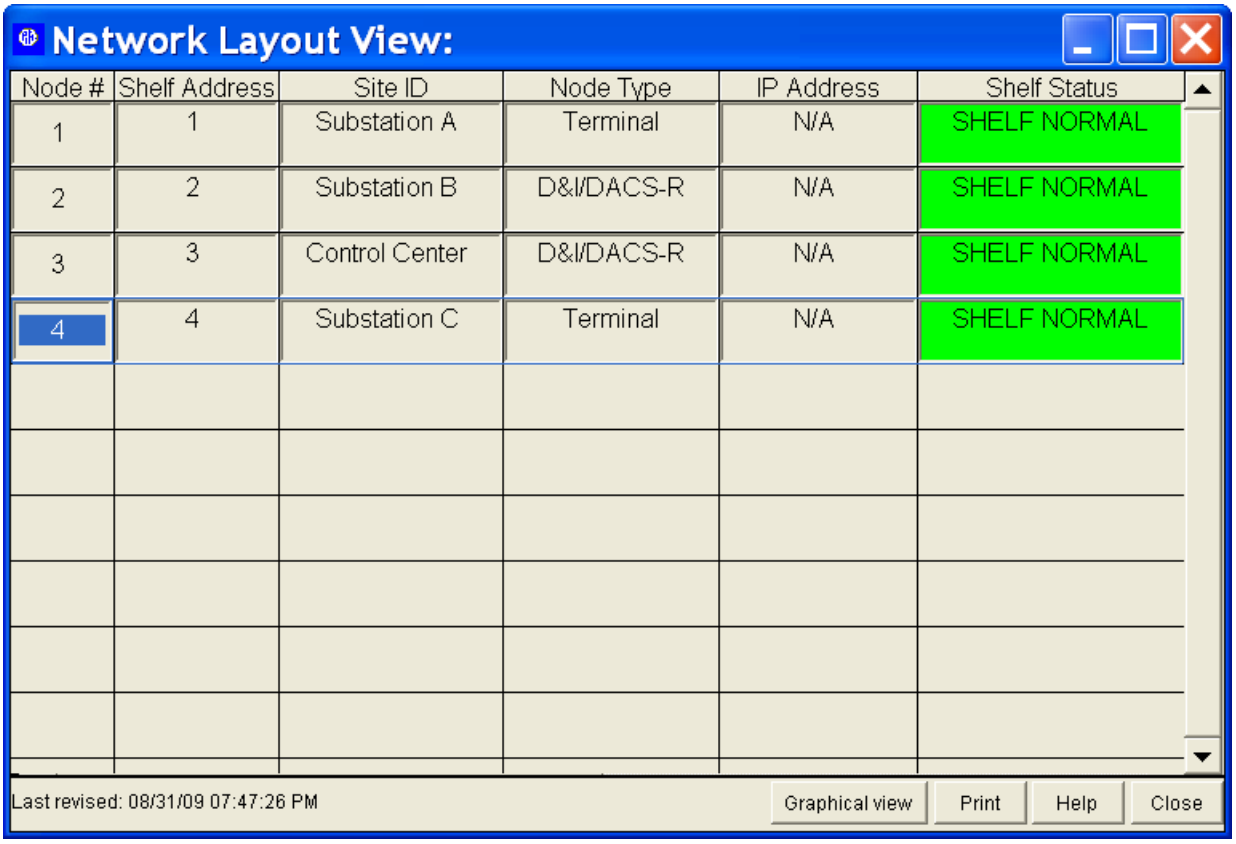

Figure 4. Creating this alternate version of Network View was a breeze, because all the data was drawn from business objects. Only the graphical portions had to change.

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).

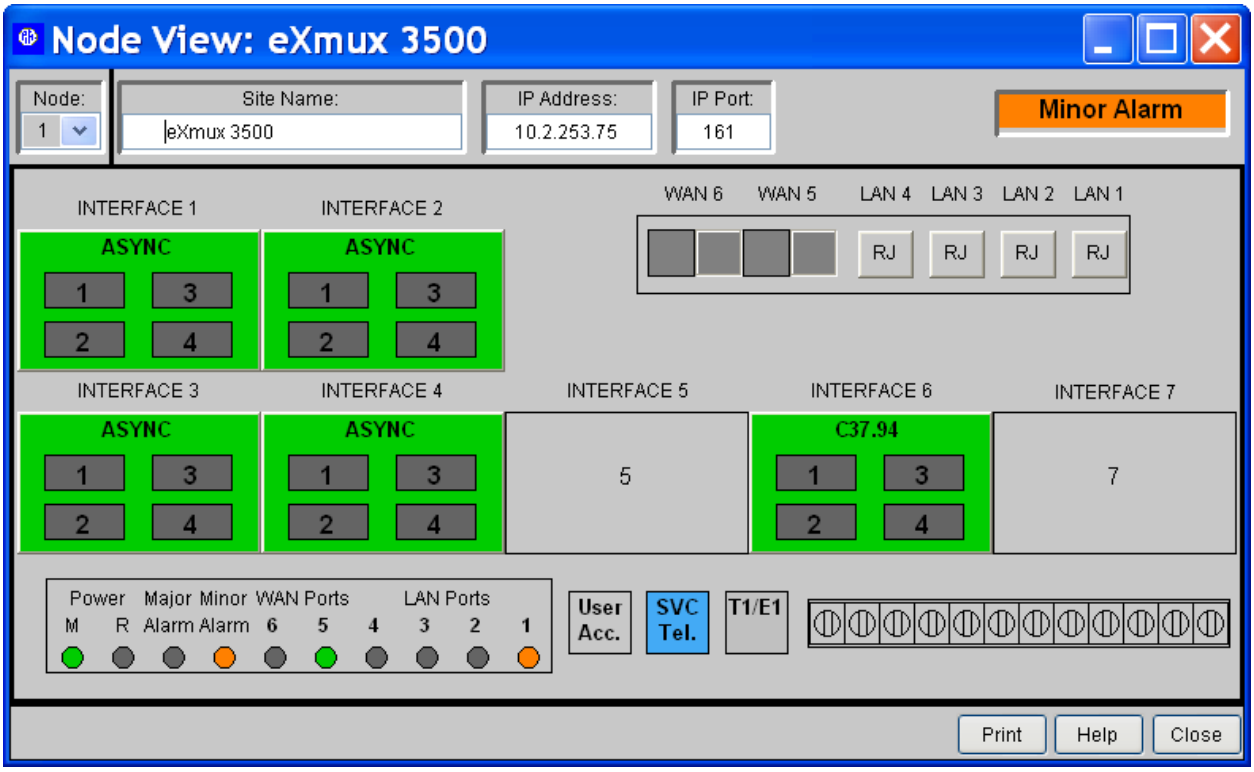

Figure 5. In the new version of NMS, Node View has changed quite a bit, but the object model underneath is nearly the same as in the older version.

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 this paper, 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.)

Now, nearly three years since 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.

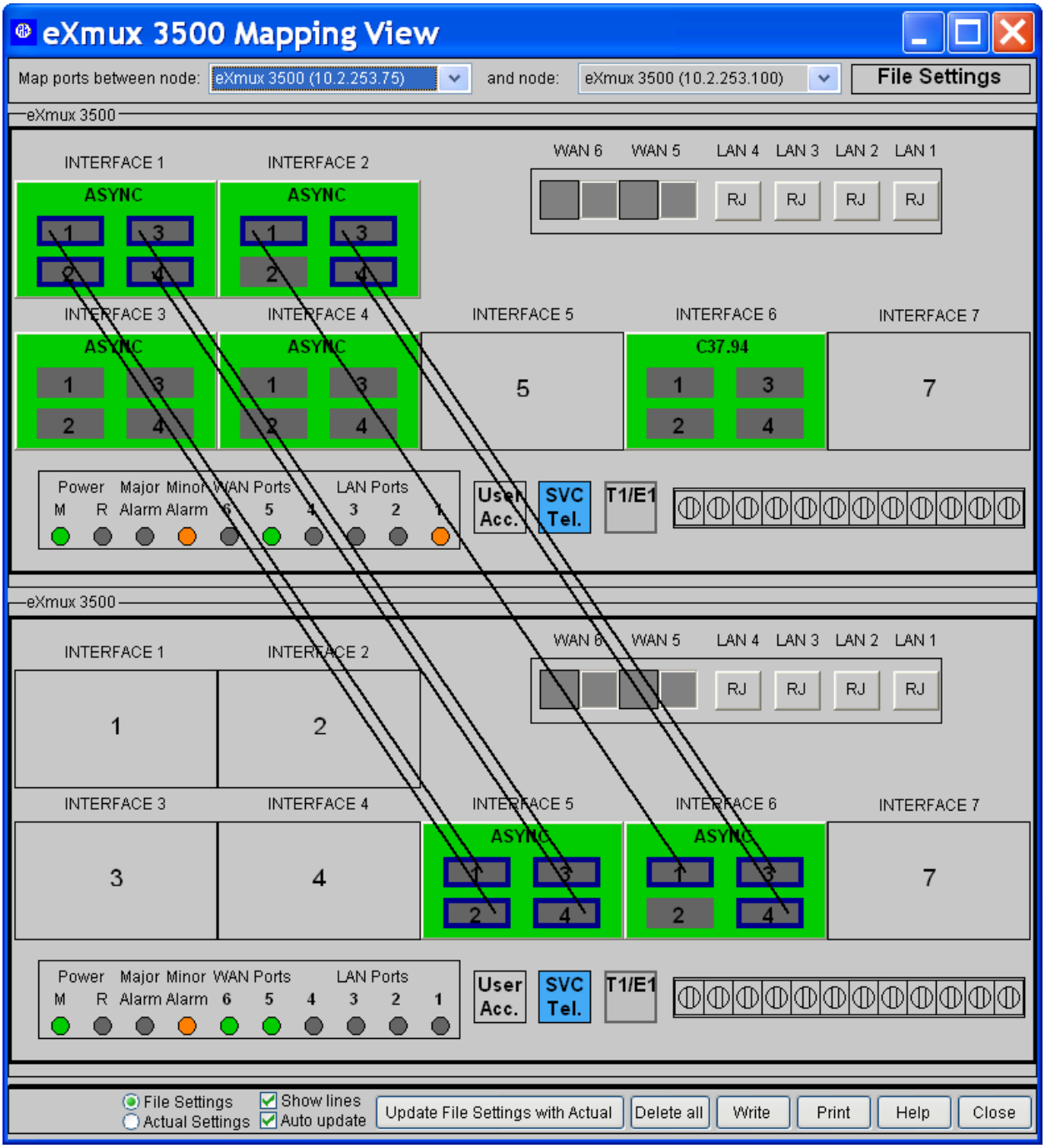

Figure 6. Mapping View is new in the most recent version of NMS. It shows the relationships between nodes. Thanks to the business objects underneath, it shares a lot of code with Node View. Both are subclassed from the same parent class.

An example: Sudoku

Of course, I can't share the code for my client's application, and even if I could, it's complex enough that it wouldn't make a good example. So, to make these ideas concrete, I've built a fairly simple Sudoku game. In case you're not familiar with it, let me introduce the game.

Sudoku is a logic puzzle. The board is typically a square divided into a grid. The grid is further subdivided into equal blocks of cells. The most common board is 9 x 9, and divided into 9 squares of 9 cells each, as in Figure 7. The figure also shows the three types of cell groups: rows, columns, and blocks.

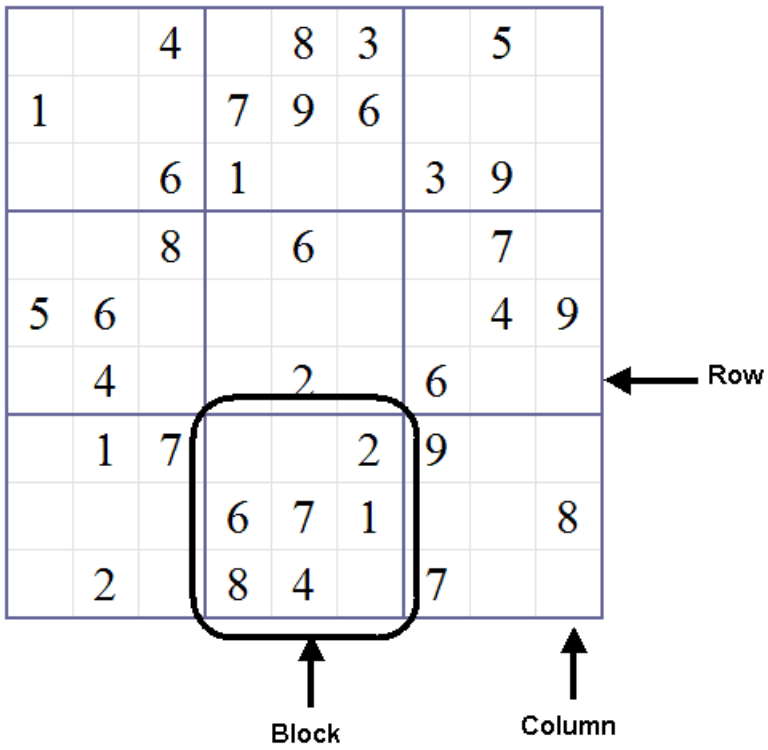

Figure 7. The standard Sudoku game features a 9x9 grid divided up into nine 3x3 squares. (Game captured from http://www.websudoku.com/.)

Initially, some subset of the cells contains numbers between 1 and the grid size (though, in fact, any set of symbols can be used). The challenge is to fill all the cells of the grid so that each row, each column and each block contains each number once. That is, for the 9 x 9 game, each row, column, and block contains all of the digits 1 through 9. Figure 8 shows the solution of the game in Figure 7. (A well-designed Sudoku has only one solution.)

9	7	4	2	8	3	1	5	6
1	5	3	7	9	6	4	8	2
2	8	6	1	5	4	3	9	7
3	9	8	4	6	5	2	7	1
5	6	2	3	1	7	8	4	9
7	4	1	9	2	8	6	3	5
8	1	7	5	3	2	9	6	4
4	3	9	6	7	1	5	2	8
6	2	5	8	4	9	7	1	3

Figure 8. The solution to the puzzle in Figure 7. Each of the digits 1 through 9 appears once in each row, column and block.

While the standard game involves a square grid divided into square blocks, variations abound. One fairly common variation is for the blocks to be non-square, simply equally sized subsets of cells, such as in Figure 9. The blocks are indicated by the darker lines, and the rules are the same: each number appears once in each row, once in each column and once in each block. (The solution to this puzzle is left as an exercise for the reader.)

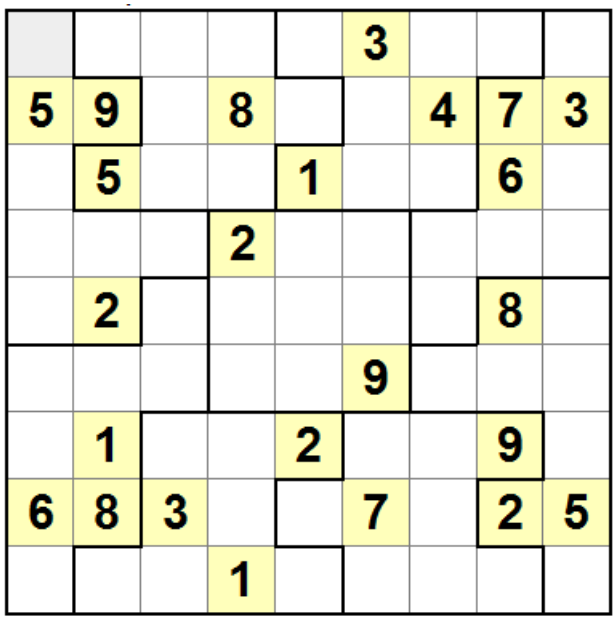

Figure 9. One Sudoku variation retains the overall square shape, but the blocks are not squares. In this version, the rules are still that each number from 1 to 9 must appear once in each row, once in each column and once in each block indicated by the darker lines. (Jigsaw Sudoku captured from http://www.sudokusplashzone.com)

The version I've implemented requires the game to be square, but supports this variation. It also supports variation of the grid size and the symbols used in the cells (with the numbers 1 to grid size as the default).

Designing business objects

Clearly, the key to making business objects useful is to have the right business objects. So how do you figure out what business objects you need?

Two key ideas guide this process. The first is that a business object represents something real (though "real" can be somewhat abstract). The second idea, though, is that business objects can have hierarchical relationships and the object model should support those relationships.

For my client's application, we have business objects representing the network as a whole, each node, each shelf, each card, and each setting. These are combined into various collections, so that you can start with the network object and traverse the entire network just by walking through the collections. Figure 10 shows (a simplified version of) the hierarchy of objects and collections.

For the Suduko game, it was easy to identify two objects. We need a game object to represent the entire game, and a cell object to represent a single cell of the grid. In between those two is where things get tricky, but also where the idea of business objects really shows its value.

Each cell in Suduko is part of three groups: a row, a column and a block. While the physical layout for each kind of group is different (and, in fact, the physical layout of a block can vary), the three kinds contain the same number of items and follow the same rules. Clearly, implementing them with common code (that is, a class) makes sense. This leads to a group business object.

Finally, there are operations needed for the entire set of rows or the entire set of columns or the entire set of blocks. This leads to one more business class, the set of groups. Figure 11 shows the hierarchy of business objects.

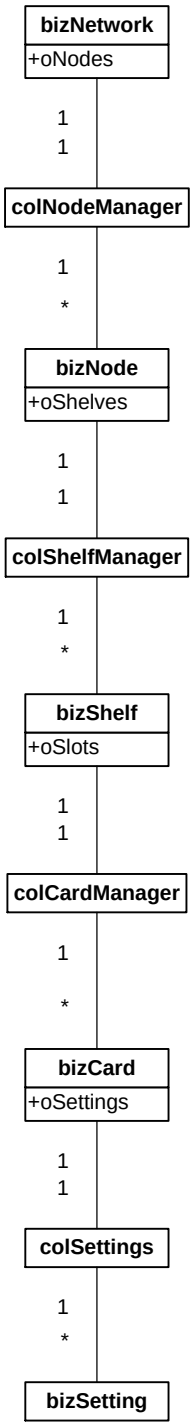

Figure 10. The business object hierarchy for the Network Management System features alternating scalar objects and collections.

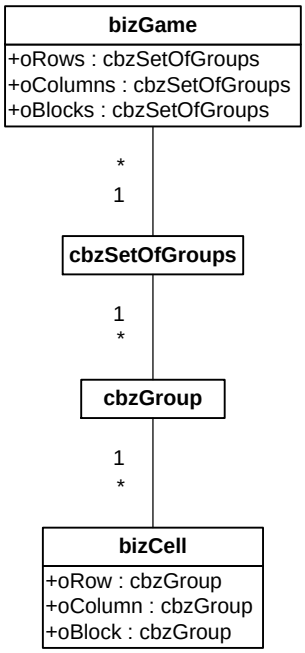

Figure 11. Like the NMS hierarchy, The object hierarchy for Sudoku uses a mix of collections and scalar objects.

Implementing business objects in VFP

As I said in "The problem with the usual examples," the earliest implementations of business objects in VFP were based on visual classes such as Container. Later, people realized that you couldn't separate implementation from interface if your business objects were interface objects, and began to base their business objects on non-visual classes. Because they're extremely lightweight classes (that is, don't use much memory), some people chose to base their business classes on the Line or Relation base classes.

I prefer to make my design decisions more transparent, so I build business objects from the Custom and Collection base classes. To date, I haven't found that I need much common functionality across business objects for graphical applications, so my "base" business object is simply a subclass of my "base" subclass of Custom. (That is, cusBase subclasses Custom, and cusBizObj subclasses cusBase.) For Sudoku, I subclassed the game and cell classes from cusBizObj; they're named bizGame and bizCell, respectively.

To build hierarchies of business objects, I use collections (introduced in VFP 8). Like arrays, collections let you aggregate objects so that you can work with the whole group or individual items. But collections function much more naturally in VFP than arrays. While there's no way to create an empty array, it's easy to have an empty collection. In addition, collections have properties and methods that make it easy to work with them. The Count property tells you how many items are in the collection, and the Add and Remove methods let you add and remove members. Collections also allow you to assign a "key" to each item to facilitate easy look-ups.

As with Custom-based business objects, I've subclassed my "base" collection class (colBase) to create a "base" collection-based business class (colBizObj). For Sudoku, I subclassed colBizObj for the group (cbzGroup) and set of groups (cbzSetOfGroups).

I should note that the idea of considering a collection class as a business class is new to me (and, I think, reflects my deepening understanding of and comfort with business classes). In my client's application, all the business objects themselves descend from Custom; then, a group of collection subclasses are used to link them together. There, the collections function as "manager" objects, and are mostly named to reflect that role.

Storing data in business objects

The whole idea of a business object is that it stores the data necessary to represent some "thing." So, you add custom properties to a business object to store that data. In NMS, for example, bizNode, the node business object has properties for various information about the node, such as its address within the network and its name. The card object, bizCard, includes properties for the type of card (circuit board) and the slot where it's stored; in the new version, where one card can stretch over two slots, it also has a property to indicate the number of slots it occupies.

In the Sudoku example, the bizCell object represents one cell in the game, so its custom properties include nRow and nColumn to indicate which cell this is in the grid. It also has nValue that contains its current value and lFixed to indicate whether the value is fixed, that is, whether it's one of the values specified at the start of the game. bizGame has an nSize property to specify the grid size; an assign method for that property ensures that any new value is a perfect square, since that's one of the rules for Sudoku. (Actually, there are some variations where that's not a rule, but this implementation doesn't handle those.) The collection-based cbzGroup has nPosition that indicates its position within its set of groups.

Linking business objects together

Custom properties are also used to represent the relationships among business objects. I find that these properties are often object references to collections. For example, in NMS, bizNode has an oShelves property that points to a collection of shelves (class colShelfManager). That class then contains the right number of bizShelf objects. bizShelf has an oSlots property that points to a collection of cards (class colCardManager—by the time I realized that the class name and the property name should match better, we'd written and tested a lot of code, so didn't change it).

In the Sudoku game, bizGame has references to three collections of class cbzSetOfGroups: oBlocks, oColumns, oRows. Because cbzSetOfGroups is subclassed from a collection class, there's no need to add properties to it to point to the individual cbzGroup objects; adding them to the collection is sufficient. The same is true for cbzGroup; because it's a collection, no custom properties are needed to track the individual cells in the group. (I'll show how the items get into the groups later in this paper.)

I've also found that backward references up the containership chain tend to be useful. So in NMS, bizCard has an oNode property that points directly to the containing bizNode object, bizNode has oNetwork to point to the bizNetwork object, and so forth.

For Sudoku, bizCell has references to the containing block, column and row objects (all based on cbzGroup). I haven't found a reason so far to give cbzGroup or cbzSetOfGroups backward references.

Backward references, though, do mean that care has to be taken when destroying these objects. If these references aren't cleared (nulled), it's possible to leave objects dangling in memory. A custom method, CleanUpReferences, high in the inheritance chain (in cusBase and colBase for Sudoku) provides a place to put the relevant code. Collections need to make sure that the method gets called for each member, so colBase.CleanUpReferences contains the code in Listing 1.

Listing 1. Every member of a collection needs to clean up its own backward references, so the custom CleanUpReferences method of colBase calls the relevant method of every member.

```
* Call all contained objects to clean up.
* This is generic code. Subclasses will need to
* null specific properties.

LOCAL oItem
FOR EACH oItem IN This FOXOBJECT
   IF PEMSTATUS(oItem, "CleanUpReferences", 5)
      oItem.CleanUpReferences()
   ENDIF
ENDFOR
```

The code needed to clean up references in scalar (non-collection) objects depends on the object. For bizCell, the code simply nulls the three backward references (Listing 2).

Listing 2. When the object model contains both forward and backward object references, it's important to clean up before destroying the objects. This code, in bizCell.CleanUpReferences, ensures that bizCell objects can be destroyed.

```
This.oRow = .null.
This.oColumn = .null.
This.oBlock = .null.
```

To ensure that this clean-up happens for each object in the hierarchy, the top object needs to start the process. So bizGame.CleanUpReferences() contains the code in Listing 3.

Listing 3. Cleaning up object references need to propagate downward. This code in bizGame.CleanUpReferences starts things off.

```
This.oRows.CleanUpReferences()
This.oColumns.CleanUpReferences()
This.oBlocks.CleanUpReferences()
```

The Destroy method of bizGame calls the class's CleanUpReferences method. Since Destroy proceeds from the container to the contained objects, this ensures that all the references have been cleaned up by the time Destroy fires for the inner objects.

Adding functionality

The second key element of a business class is a set of methods to provide operations on the data. What methods you need obviously depends on what the object represents. Typically, the methods you add fall into several categories: building and destroying the object model, storing and retrieving data, retrieving objects contained in this object, querying data of this object and manipulating data in this object. (While the sections that follow show code for many methods from the Sudoku game, not every method is shown here.)

Building and destroying the object model

One set of methods lets you manage the object hierarchy itself. These methods add and remove objects, and create the connections among them. When adding items to collections, I like to assign keys, so that I can easily find particular members of the collection. It's also not unusual for these methods to delegate tasks down the object hierarchy.

In NMS, for example, bizShelf has an AddCard method that's called both when initially constructing the object hierarchy from stored data and when the user adds a card through the user interface. bizShelf doesn't actually do the work, though; it calls the AddCard method of its oSlots collection (colCardManager class). In fact, in this case, delegation can come from even farther up the hierarchy. bizNode also has an AddCard method. It delegates to the appropriate bizShelf object.

The colCardManager.AddCard method figures out whether the specified card can be added in the specified slot and, if so, creates a bizCard object and adds it to the oSlots collection. The slot number is converted to character and used as the key for the object, making it easy to request the card in a particular slot.

The object model for the Sudoku game is much simpler (as are the requirements for managing the objects since once you build the object hierarchy for a particular game, its structure doesn't change), but it follows the same principles. bizGame has a method called SetupGame (Listing 4), which instantiates the three cbzSetOfGroups objects. It then creates all the necessary bizCell objects and assigns them to the appropriate groups.

Listing 4. bizGame's custom SetupGame method constructs the objects needed to represent the Sudoku data.

```
LPARAMETERS nSize

LOCAL nRow, nColumn, oCell

IF VARTYPE(m.nSize) = "N"
   This.nSize = m.nSize
ENDIF
```

```
* Create the sets of groups
This.oRows = NEWOBJECT("cbzSetOfGroups", "bizobjs", "", This.nSize)
This.oColumns = NEWOBJECT("cbzSetOfGroups", "bizobjs", "", This.nSize)
This.oBlocks = NEWOBJECT("cbzSetOfGroups", "bizobjs", "", This.nSize)

* Create cells and add them to the right groups
FOR nRow = 1 TO This.nSize
   FOR nColumn = 1 TO This.nSize
      oCell = NEWOBJECT("bizCell", "bizobjs")
      WITH oCell
         .nRow = m.nRow
         .nColumn = m.nColumn
      ENDWITH

      * Add it to the right row, column and block
      This.oRows.AddCell(m.oCell, m.nRow, m.nColumn, "R")
      This.oColumns.AddCell(m.oCell, m.nColumn, m.nRow, "C")

      * Call a method to handle the block so we can subclass
      * to handle variants
      This.AddCellToBlock(m.oCell, m.nRow, m.nColumn)
   ENDFOR
ENDFOR
```

The Init method of cbzSetOfGroups (shown in Listing 5) handles creation of the individual cbzGroup objects needed for each set. It receives the number of groups as a parameter (for the standard 9x9 Sudoku game, nGroups is 9) and adds that many groups to the collection. The position of the object within the group (that is, the order of creation) converted to character is used as the key to the collection.

Listing 5. Each set of groups, represented by a cbzSetOfGroups object, needs one cbzGroup object

```
LPARAMETERS nGroups

* Add the specified number of groups to this set, using
* the group number as the key for each.

LOCAL nGroup, oGroup, cKey

FOR nGroup = 1 TO m.nGroups
   oGroup = NEWOBJECT("cbzGroup", "bizObjs")
   oGroup.nPosition = m.nGroup
   cKey = TRANSFORM(m.nGroup)
   This.Add(m.oGroup, m.cKey)
ENDFOR
```

To add each bizCell object to the appropriate row or column, bizGame.SetupGame calls the AddCell method of oRows and oColumns, which are both based on cbzSetOfGroups. AddCell receives four parameters: the bizCell object, the number of the group to which the cell is to be added, the position in the group for that cell, and the type of group (row, column or

block) we're dealing with. The method, shown in Listing 6, finds the right group and calls its AddCell method, passing along the remaining parameters.

Listing 6. cbzSetOfGroups.AddCell figures out which group a cell is to be added to and calls that group's AddCell method.

```
LPARAMETERS oCell, nGroup, nPosInGroup, cGroupType

LOCAL oGroup, cKey, lSuccess

oGroup = This.GetGroup(m.nGroup)

lSuccess = .F.
IF NOT ISNULL(m.oGroup)
   lSuccess = oGroup.AddCell(m.oCell, m.nPosInGroup, m.cGroupType)
ENDIF

RETURN m.lSuccess
```

cbzGroup's AddCell method adds the bizCell object to the collection (itself), using the desired position in the group (converted to character) as the key. It also calls the cell's SetBackPointer method to set the cell's backward pointer to the group; the cGroupType parameter determines which of the backward pointers is set. cbzGroup.AddCell is shown is Listing 7, while the bizCell.SetBackPointer is in Listing 8.

Listing 7. The actual work of adding the cell to the group happens in cbzGroup.AddCell.

```
LPARAMETERS oCell, nPosition, cGroupType

LOCAL cKey

cKey = TRANSFORM(m.nPosition)
This.Add(m.oCell, m.cKey)

oCell.SetBackPointer(m.cGroupType, This)

RETURN
```

Listing 8. bizCell's SetBackPointer method uses the cGroupType parameter to figure out which of the backward pointers to set, and then points that to the group to which the cell was just added.

```
LPARAMETERS cGroupType, oGroup

DO CASE
CASE m.cGroupType = "R"
   This.oRow = m.oGroup

CASE m.cGroupType = "C"
   This.oColumn = m.oGroup

CASE m.cGroupType = "B"
   This.oBlock = m.oGroup

OTHERWISE
```

```
   * Should never happen
ENDCASE
```

Adding a cell to the right block is a little trickier, because the definition of a block can vary. So another method of bizGame, AddCellToBlock, is used. In bizGame, this method (Listing 9) uses the standard Sudoku rules, dividing the grid into squares whose size is the square root of the overall grid's size. (That is, for a 9x9 game, each block is 3x3.) Arbitrary, the blocks are numbered from left to right, row by row, and cells within the blocks are numbered the same way. To handle variants, bizGame must be subclassed.

Listing 9. The AddCellToBlock method of bizGame uses standard Sudoku rules to add the bizCell to the right block.

```
* Add a cell to the right block. This code
* uses standard Sudoku rules. Subclass and replace
* this code for variants.
LPARAMETERS oCell, nRow, nColumn

LOCAL nBlock, nPosInBlock, nBlockSize

nBlockSize = SQRT(This.nSize)

nBlock = INT((m.nRow-1)/m.nBlockSize) * m.nBlockSize + ;
         INT((m.nColumn-1)/m.nBlockSize) + 1
nPosInBlock = MOD(m.nRow-1, m.nBlockSize) * m.nBlockSize + ;
              MOD(m.nColumn-1, m.nBlockSize) + 1
This.oBlocks.AddCell(m.oCell, m.nBlock, m.nPosInBlock, "B")

RETURN
```

For cleaning up the object hierarchy, you generally need something like the CleanUpReferences method described in "Linking business objects together" earlier in this document.

Retrieving and storing data

You usually need methods that retrieve data from a data source and that store the data before destroying the object hierarchy. The details of what methods are needed vary with the application.

NMS stores network data in a set of tables. (It's actually a little more complicated than that. The new version of NMS stores network data as XML, then reads the XML into a set of tables before converting it to objects.) The business objects need methods that read the data from those tables and create and populate the appropriate objects. They also need methods to store the data in the objects back into the tables. So, bizNetwork has a ReadNetwork method that stores the network level data in the appropriate properties, and then calls the ReadNodes method of its oNodes collection (based on colNodeManager). ReadNodes processes the table of nodes, creating a bizNode object and adding it to the collection for each node, then calling the new node's ReadNode method. This process continues all the way down the containership hierarchy so that all the data for the network

is added to the object hierarchy. There's a corresponding set of methods (WriteNetwork, WriteNodes, WriteNode, etc.) that write data back from the objects to the tables.

The Sudoku game has much simpler needs for data retrieval and storage. There's no data to store between sessions. There is a need, though, to load data for a game.

My initial design for game data uses a comma-separated text file for each game, containing one line for each fixed value. Each line is in the form: row, column, value. bizGame has a method, AddFixedData (shown in Listing 10), that accepts a string in the format of that file, and parses it to populate the game with its initial data. In this version of the game, the code that instantiates the game reads the text file into a string and passes it along to this method.

Listing 10. bizGame's AddFixedData method accepts a string with the game data and sets up the fixed values.

```
* Fill in the fixed values for thix game.
* These are provided as a text string with one value per line.
* Each line is a comma-separated triple, giving the row, the column, and the value.
* So, for example:
*   1,3,4
*   2,7,1
* would indicate that row 1, column 3 contains 4 and row 2, column 7 contains 1.

LPARAMETERS cFixedData

LOCAL aFixedValues[1], nValueCount, nValue, aOneLine[1], nDataItems

nValueCount = ALINES(m.aFixedValues, m.cFixedData)

FOR nValue = 1 TO m.nValueCount
   nDataItems = ALINES(m.aOneLine, m.aFixedValues[m.nValue], ",")
   IF m.nDataItems = 3
      * Process this line
      This.oRows.SetValue(m.aOneLine[1], m.aOneLine[2], VAL(m.aOneLine[3]), .T.)
   ELSE
      * Skip this line
   ENDIF
ENDFOR

RETURN
```

After the game was working, I realized that this format for the data limited the game's functionality. "Handling Changed Requirements" later in this document describes how I introduced a new file format and additional functionality into the object model.

Retrieving objects

Business object code (or the code that uses the business objects) often needs to find another particular object, so business objects that contain other business objects typically have methods to retrieve specific member objects. I generally give such methods a name beginning with "Get."

In NMS, the bizNetwork object has several methods for retrieving a particular node—one looks it up by node number, another by its unique identifier, and a third by its address in the network. bizNode has a number of methods for retrieving a particular card, as well as a method for retrieving a shelf. Other objects have similar methods.

The Sudoku game's bizGame object has a method called GetCell to retrieve a cell based on its row and column; it's shown in Listing 11. It uses retrieval methods from two objects lower in the hierarchy. Using those methods prevents bizGame from knowing too much about the structure of the objects it contains. The internal structure of cbzSetOfGroups or cbzGroup can change without breaking this method, as long as the GetGroup and GetCell methods continue to return the specified group and cell objects, respectively.

Listing 11. The GetCell method of bizGame retrieves a bizCell object based on its row and column.

```
LPARAMETERS nRow, nCol

LOCAL oGroup, oCell

oGroup = This.oRows.GetGroup(m.nRow)
IF NOT ISNULL(m.oGroup)
   oCell = oGroup.GetCell(m.nCol)
ELSE
   oCell = .null.
ENDIF

RETURN m.oCell
```

cBzSetOfGroups.GetGroup retrieves a particular member of the collection, based on its position. This method (shown in Listing 12) takes advantage of the key assigned to each group within a set.

Listing 12. GetGroup retrieves a single group from within the set, based on its position. Position is used as the key when groups are added to the collection.

```
* Return the specified group from this set.
LPARAMETERS nGroup

LOCAL cKey, oGroup

cKey = TRANSFORM(m.nGroup)

TRY
   oGroup = This.Item[m.cKey]
CATCH
   oGroup = .null.
ENDTRY

RETURN m.oGroup
```

The GetCell method of cbzGroup (Listing 13) accepts a position as parameter and returns the cell in that position in the group. GetGroup and GetCell have the same structure; each

attempts to grab the member of the collection with a particular key. TRY-CATCH handles the possibility that there is no such member; in that case, the method returns .null.

Listing 13. cbzGroup.GetCell retrieves the bizCell in a specified position.

```
* Get the specified cell from this group.
LPARAMETERS nCell

LOCAL cKey, oCell

cKey = TRANSFORM(m.nCell)
TRY
   oCell = This.Item[ m.cKey ]

CATCH
   oCell = .null.

ENDTRY

RETURN m.oCell
```

Querying data

In my experience, business objects tend to have lots of methods to answer questions about their status. Many of these methods are wrappers around fairly simple conditions; as with "Get" methods for object retrieval, using methods instead of putting the conditions right into the code makes it safer to make changes to the objects being queried. Query methods often have names beginning with "Is" or "Can." (Some of these methods implement business rules, letting you know whether the rules have been followed or not.)

In NMS, for example, bizNode has a method called IsOnline that returns a value indicating whether the node is currently online. bizCard has several query methods, including a few to determine whether the object represents a particular kind of card.

The Suduko game uses a number of query methods as well. bizCell has two, IsEmpty and IsFixed. IsEmpty returns .T. if the nValue property is set to 0, that is, if no valid value has been assigned to the cell. IsFixed returns the value of the cell's lFixed property.

cbzGroup has a whole set of "Is" methods. They include IsFull (Listing 14), which checks whether all cells have been assigned a value, and IsValid (Listing 15), which checks whether the set of values in this group is valid according to the rules of the game.

Listing 14. cbzGroup's IsFull method checks whether all cells in the group have values.

```
LOCAL lReturn, oCell

lReturn = .T.
FOR EACH oCell IN This FOXOBJECT
   IF oCell.IsEmpty()
      lReturn = .F.
```

```
      EXIT
   ENDIF
ENDFOR

RETURN m.lReturn
```

Listing 15. The IsValid method of cbzGroup checks to see whether the values in the group's cells are unique.

```
LOCAL oCell, lReturn, aValuesUsed[This.Count]

* To determine whether the set of values in this group
* is valid, loop through. For each cell, if it's not empty,
* check the corresponding array element. If it's .T., then we previously
* found this value, so the group is not valid. If the array element
* is .F., then set it to .T. to indicate that we've seen
* this value.

lReturn = .T.
FOR EACH oCell IN This FOXOBJECT
   IF NOT oCell.IsEmpty(m.oCell)
      IF aValuesUsed[oCell.nValue]
         lReturn = .F.
         EXIT
      ELSE
         aValuesUsed[oCell.nValue] = .T.
      ENDIF
   ENDIF
ENDFOR

RETURN m.lReturn
```

The IsComplete method (Listing 16) uses IsFull and IsValid to figure out whether this group has a complete set of values (implementing the business rule about what constitutes a complete group). cbzGroup has several other query methods as well.

Listing 16. cbzGroup.IsComplete checks whether the group has a complete set of values.

```
RETURN This.IsFull() AND This.IsValid()
```

cbzSetOfGroups has only one query method; IsComplete checks each group in the set for completeness. It's shown in Listing 17. Again here, using a method of the contained object (bizGroup) rather than performing the checks directly means that the structure and behavior of bizGroup can change without breaking this method.

Listing 17. The IsComplete method of cbzSetOfGroups uses cbzGroup.IsComplete to check each group in the set.

```
LOCAL oGroup, lReturn

lReturn = .T.

FOR EACH oGroup IN This FOXOBJECT
   lReturn = m.lReturn AND oGroup.IsComplete()
ENDFOR
```

```
RETURN m.lReturn
```

bizGame also has an IsComplete method, shown in Listing 18. It checks each of the three sets of groups (the rows, the columns and the blocks) for completeness.

Listing 18. The bizGame-level IsComplete method checks the rows, columns and blocks for completeness.

```
LOCAL lReturn

lReturn = This.oRows.IsComplete()
IF m.lReturn
   lReturn = This.oColumns.IsComplete()

   IF m.lReturn
      lReturn = This.oBlocks.IsComplete()
   ENDIF
ENDIF

RETURN m.lReturn
```

Query methods don't have to just return a logical value; they can also assemble data that answers the question. bizGame has a pair of methods (CheckForConflicts and CheckGroupsForConflicts) that populates a collection with a list of cells containing values that are, in some way, in conflict with other data in the grid. CheckForConflicts is shown in Listing 19 and CheckGroupsForConflicts in shown in Listing 20. CheckGroupsForConflicts calls the GetConflicts method of cbzGroup (Listing 21) for each group in the set to retrieve a collection of conflicts for that group. Like the IsComplete methods, these methods implement a set of business rules.

Listing 19. bizGame's CheckForConflicts method creates a collection of cells that are in conflict with other data in the grid.

```
LOCAL oConflicts, oGroupConflicts, oRow, oColumn, oBlock, oConflict

* Call lower-level method to do the actual checking
* and accumulate the results

oConflicts = CREATEOBJECT("Collection")

This.CheckGroupsForConflicts(This.oRows, oConflicts)
This.CheckGroupsForConflicts(This.oColumns, oConflicts)
This.CheckGroupsForConflicts(This.oBlocks, oConflicts)

RETURN m.oConflicts
```

Listing 20. The CheckGroupsForConflicts method of bizGame figures out which set of groups to look at, then loops through the groups in that set, retrieving a collection of conflicts from each and adding those to the result.

```
LPARAMETERS oGroupsToCheck, oConflicts

LOCAL oGroup, oGroupConflicts, oConflict, cKey
```

```
FOR EACH oGroup IN oGroupsToCheck FOXOBJECT
   oGroupConflicts = oGroup.GetConflicts()
   FOR EACH oConflict IN oGroupConflicts FOXOBJECT
      cKey = "R" + TRANSFORM(oConflict.nRow) + "C" + TRANSFORM(oConflict.nColumn)
      IF oConflicts.GetKey(m.cKey) = 0
         oConflicts.Add(m.oConflict, m.cKey)
      ENDIF
   ENDFOR
ENDFOR

RETURN
```

Listing 21. cbzGroup's GetConflicts method returns a collection listing the cells in the group that conflict with other data.

```
* Return a collection of cells in this group that are in
* conflict. To be in conflict means that with their
* current values, the cells are not consistent with the
* group being valid. Fixed cells are always valid, so
* when a varying cell and a fixed cell have the same value,
* only the varying cell will be included in the return.

LOCAL oConflicts, oCell, aValueCount[This.Count], nValue

oConflicts = CREATEOBJECT("Collection")

FOR nValue = 1 TO This.Count
   aValueCount[m.nValue] = 0
ENDFOR

* Need to make two passes. In pass 1, count the usage
* for each value. In pass 2, make the list of conflicts.
FOR EACH oCell IN This FOXOBJECT
   IF NOT oCell.IsEmpty()
      aValueCount[oCell.nValue] = aValueCount[oCell.nValue] + 1
   ENDIF
ENDFOR

FOR EACH oCell IN This FOXOBJECT
   IF NOT oCell.IsEmpty()
      IF aValueCount[oCell.nValue] > 1 AND NOT oCell.IsFixed()
         oConflicts.Add(m.oCell)
      ENDIF
   ENDIF
ENDFOR

RETURN m.oConflicts
```

Manipulating data

Most business objects need methods that manipulate their own data. This is where the action takes place. These methods let you change data based on user actions (or other actions—in NMS, some data changes reflect changes occurring on the actual network hardware).

In NMS, one of the key concepts is that a node can be running and responsive ("online"), not responding ("offline") or in a maintenance mode ("forced offline"). So bizNode has a method SetOnlineStatus that changes the node's cStatus property to reflect its current status.

The Sudoku game's bizCell class has a SetValue method (Listing 22), called to change the number assigned to a particular cell, clearly the key action in this game. The method ensures that the cell can be changed (that is, that it's not fixed) and then, if permitted, assigns the new value. The same method is used during both set-up and game play, so it accepts an optional lFixed parameter, used to set the lFixed method. bizGame's AddFixedData method passes .T. for this parameter, in order to set up the initial set of values.

Listing 22. bizCell's SetValue method changes the number assigned to a cell.

```
* Set this cell to the specified value. If lFixed
* is passed and true, mark this value as fixed.

LPARAMETERS nValue, lFixed

LOCAL lSuccess
* First, check whether this cell is already fixed.
IF This.IsFixed()
   lSuccess = .F.
ELSE
   This.nValue = m.nValue
   This.lFixed = m.lFixed
ENDIF

RETURN m.lSuccess
```

cbzSetOfGroups and cbzGroup also have SetValue methods that figure out which object to talk to and delegate the operation down to that object.

Connecting business objects to the user interface

The methods of the business objects form the engine of the application. If you want, you can use them programmatically (or even from the Command Window) to perform all the operations. But a user interface to allow users to interact with the objects makes it much easier. You then need some way to connect the business objects to the controls in the UI. Once you establish such a connection, forms and controls can call on business object methods to take appropriate action.

A conventional data entry application might bind controls to properties of the business objects. Alternatively, data might be stored in a cursor, with the business objects simply providing operations on the data.

For a highly graphical application like NMS or Sudoku, however, there may not even be controls to which much of the data can be directly bound, as much of the UI may be built

from objects without ControlSources. However, UI objects can have references to business objects, in order to call on the business objects for data and services.

For example, in NMS, the Network View form has a custom property, oNetwork, that points to the bizNetwork object. Similarly, the Node View form has an oNode property that points to the bizNode object for the node currently displayed. The class cntShelf, which provides the visual representation of one shelf, has an oBizShelf property that points to the corresponding bizShelf object.

Sudoku uses a container class, cntBoard, to hold the grid. It has an oBizGame property that is assigned an object reference to the corresponding bizGame object. Individual cells are represented by cntCell objects, which have a custom oBizCell property to point to the corresponding bizCell.

The game also has a form class that contains methods to instantiate cntBoard and to call on both UI and business object methods to do things such as start a new game, or clear all the user-entered data out from the current game, allowing the user to start over. Figure 12 shows the game board.

Constructing the forms

In conventional applications, forms are generally pretty much defined at design-time. In highly graphical applications, much form construction happens at runtime. The graphical objects have methods for constructing themselves based on their business objects. In NMS, the Network View form has a method called DrawNetwork that controls the process of drawing the nodes on the form. Similarly, the Node View form has a DrawNode method that converts the data for the referenced bizNode into its graphical representation. Each of these methods uses the business objects to guide construction of the appropriate graphical objects.

In Sudoku, cntBoard has a method BuildBoard (Listing 23) that receives a bizGame object as a parameter and adds cells and dividers to the board based on that object and the hierarchy it references. BuildBoard calls two additional methods, AddCells and AddDividers, to do the actual work.

Listing 23. cntBoard's BuildBoard method controls the process of filling the board with the appropriate number of cells and adding the right data.

```
LPARAMETERS oBizGame

This.oBizGame = m.oBizGame

This.nSize = This.oBizGame.nSize

This.AddCells()
This.AddDividers()

RETURN
```

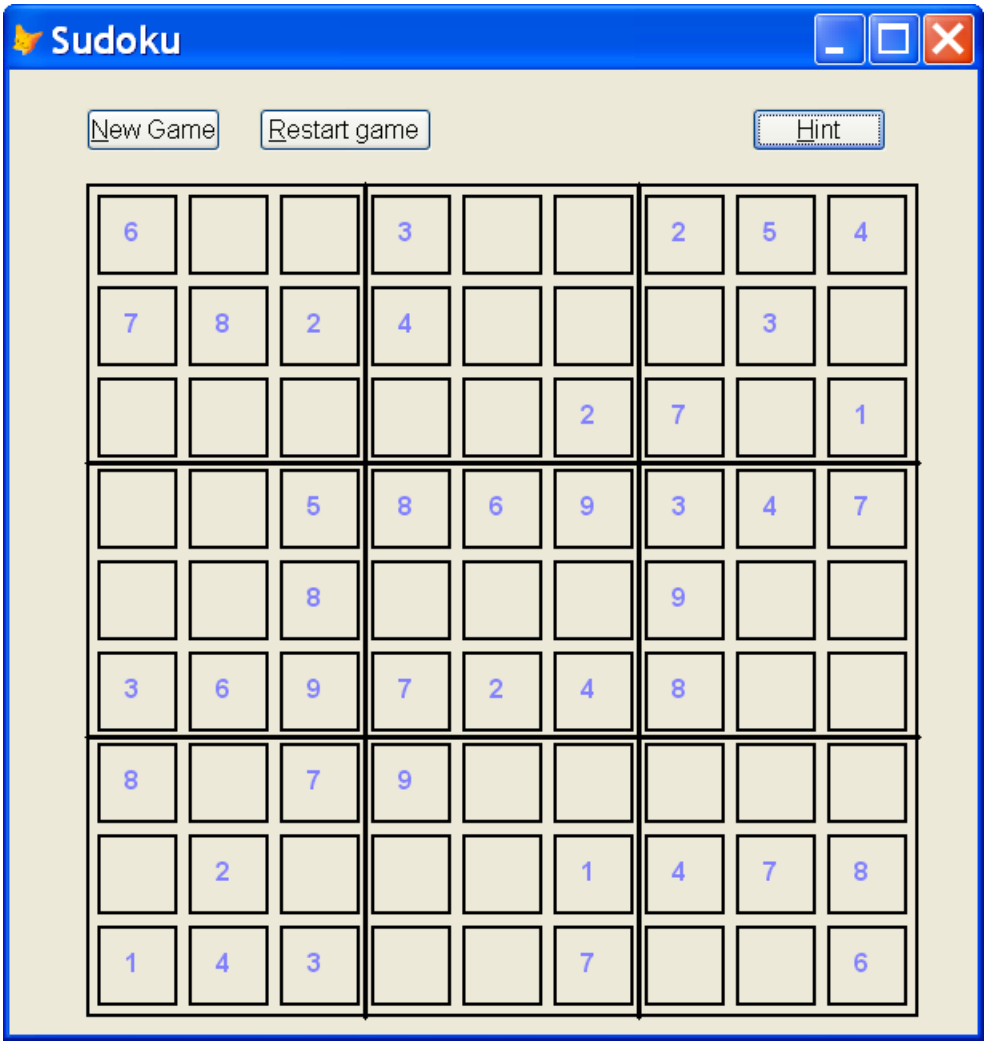

Figure 12. The Sudoku form contains a cntBoard object and a few buttons. Form code handles construction of the board, as well as calling on cntBoard and the business objects to implement the button actions.

AddCells, shown in Listing 24, loops through the required number of rows and columns, adding cells one at a time, and connecting each to the appropriate bizCell object. This method also binds the custom ValueChanged method of each cell to the board's BoardChanged method; the next section shows how this architecture lets us update the display after each user change.

Listing 24. The Sudoku cells are added to the grid by cntBoard's AddCells method, which hooks them to the right bizCell objects.

```
LOCAL nRow, nCol, cCellName, oCell, oBizCell

FOR nRow = 1 TO This.nSize
   FOR nCol = 1 TO This.nSize
      cCellName = "cntCellR" + TRANSFORM(m.nRow) + "C" + TRANSFORM(m.nCol)
      oBizCell = This.oBizGame.GetCell(m.nRow, m.nCol)

      This.NewObject(m.cCellName, "cntCell", "Components.VCX", "", m.oBizCell)
      oCell = EVALUATE("This." + m.cCellName)
```

```
      WITH oCell
         .Left = (m.nCol-1) * (oCell.Width + This.nSpaceBetween) + ;
                 This.nSpaceBetween
         .Top = (m.nRow-1) * (oCell.Height + This.nSpaceBetween) + ;
                This.nSpaceBetween
         .Visible = .T.
      ENDWITH

      * Bind changes in this cell to the board, so we can check for validity.
      BINDEVENT(oCell, "ValueChanged", This, "BoardChanged", 1)
   ENDFOR
ENDFOR

* Size the container
This.Width = This.nSize * (oCell.Width + This.nSpaceBetween) + This.nSpaceBetween
This.Height = This.nSize * (oCell.Height + This.nSpaceBetween) + This.nSpaceBetween

RETURN
```

The Init method of cntCell handles a variety of "bookkeeping" to set the cell up with the right data.

Listing 25. cntCell's Init method links the cell container to the corresponding bizCell object, and sets key properties of the container based on data in the business object.

```
LPARAMETERS oBizCell

DODEFAULT()
This.txtCellValue.Center()

This.oBizCell = m.oBizCell
IF This.oBizCell.IsFixed()
   This.txtcellvalue.ReadOnly = .T.
   This.txtCellValue.ForeColor = This.nFixedCellColor
ENDIF

IF NOT This.oBizCell.IsEmpty()
   This.SetValue(This.oBizCell.nValue)
ENDIF

RETURN
```

The version of AddDividers in cntBoard handles square blocks. To handle, non-square variations, cntBoard must be subclassed.

Listing 26. In cntBoard, AddDividers create square blocks. For variants with non-square blocks, subclass and override this method.

```
* Add block dividers.

LOCAL nBlocks, nCellHeight, nCellWidth, nBlock, oLine, cLineName

* Number of Blocks in each direction
nBlocks = SQRT(This.nSize)
```

```
IF PEMSTATUS(This, "cntCellR1C1", 5)
   nCellHeight = This.cntCellR1C1.Height
   nCellWidth = This.cntCellR1C1.Width
ELSE
   * Default
   nCellHeight = 50
   nCellWidth = 50
ENDIF

* Horizontal first
FOR nBlock = 1 TO m.nBlocks - 1
   cLineName = "linHorizonal" + TRANSFORM(m.nBlock)
   This.NewObject(m.cLineName, "linDividerHorizontal", "Components.vcx")
   oLine = EVALUATE("This." + m.cLineName)

   WITH oLine
      .Left = 0
      .Width = This.Width
      .Top = m.nBlock * m.nBlocks * (m.nCellHeight + This.nSpaceBetween) + ;
             This.nSpaceBetween/2
      .Visible = .T.
   ENDWITH
ENDFOR

* Now vertical
FOR nBlock = 1 TO m.nBlocks - 1
   cLineName = "linVertical" + TRANSFORM(m.nBlock)
   This.NewObject(m.cLineName, "linDividerVertical", "Components.vcx")
   oLine = EVALUATE("This." + m.cLineName)

   WITH oLine
      .Left = m.nBlock * m.nBlocks * (m.nCellWidth + This.nSpaceBetween) + ;
              This.nSpaceBetween/2
      .Top = 0
      .Height = This.Height
      .Visible = .T.
   ENDWITH
ENDFOR

RETURN
```

Calling on business objects

Once the visual representation of the business objects has been constructed, whether it's conventional or highly graphical, the user can interact with it. As a user acts, the form needs to respond. Many responses call on the underlying business objects for things such as checking validity, adding graphical objects, storing data, and much more.

For example, in NMS, a user can right-click in Network View and add a node to the network. Doing so adds a bizNode object to the collection (after collecting additional information from the user) and then to the graphical display. In the newer version of Node View, the ports (the numbered squares in Figure 5) have tooltips that provide information about how

they're connected to ports of other nodes. The code for those tooltips calls on the underlying business objects to provide the data to display.

In Sudoku, the cntCell object contains a textbox. Its Valid method uses RaiseEvent() to fire the container's custom Valid method (shown in Listing 27). That method checks the user's input for basic validity (that is, whether the entry is in the set of values accepted by this game); then, it calls the cell's ValueChanged method.

Listing 27. When a user types a new value into a cell, cntCell.Valid fires.

```
IF NOT This.Parent.IsInputValid(This.txtCellValue.Value)
   This.ClearValue()
ENDIF

This.ValueChanged()
```

ValueChanged (Listing 28) converts the user's entry into a numerical value (this design supports any set of characters to fill the grid) and then tells the bizCell to update itself. As noted in the previous section, cntCell.ValueChanged is bound to cntBoard.BoardChanged, so after this method finishes, BoardChanged fires. It checks whether the new value is valid according to the rules of the game, and if so, checks whether the game is complete; it's shown in Listing 29. Once the game is complete, it prevents the user from making further changes.

Listing 28. cntCell's ValueChanged method converts the new value into a number and calls SetValue for the associated bizCell.

```
* Pass the new value back to the object model
LOCAL nValue

nValue = This.ConvertDisplayToValue(This.txtcellvalue.Value)

This.oBizCell.SetValue(m.nValue)

RETURN
```

Listing 29. The BoardChanged method of cntBoard fires after a change in any cell. If the new value is valid, it checks whether the puzzle has been completed, using oBizGame's IsComplete method.

```
* Something changed. Check for validity.
IF NOT This.lWinReported
   IF NOT This.CheckForConflicts()
      * No conflicts, so check whether we're done.
      IF This.oBizGame.IsComplete()
         IF PEMSTATUS(ThisForm, "GameIsComplete", 5)
            ThisForm.GameIsComplete()
         ENDIF
         This.FreezeBoard()
         This.lWinReported = .T.
      ENDIF
   ENDIF
ENDIF
```

```
RETURN
```

BoardChanged calls on cntBoard's CheckForConflicts method. This method uses bizGame's CheckForConflicts method to get a list of cells that have some kind of violation of the rules, and then tells each of those cells to show that it has a conflict. In this implementation, I've chosen to show conflicts by using a different forecolor, but using a separate ShowConflict method means that you can easily change the way conflicts are shown.

Listing 30. The CheckForConflicts method of cntBoard calls bizGame's CheckForConflicts method. That method returns a collection of bizCell objects that represent rules violations. The method traverses that collection and finds the corresponding cntCell for each, so that it can be told to show the conflict.

```
LOCAL oConflicts, oBizCell, oCell

oConflicts = This.oBizGame.CheckForConflicts()

IF NOT ISNULL(m.oConflicts) AND oConflicts.Count > 0
   FOR EACH oBizCell IN m.oConflicts FOXOBJECT
      oCell = This.GetCellByBizCell(m.oBizCell)
      IF NOT ISNULL(m.oCell)
         oCell.ShowConflict()
      ENDIF
   ENDFOR
ENDIF

RETURN oConflicts.Count > 0
```

Handling changed requirements

One of the big selling points for business objects is that they make it easier to change your application. You can change the engine without changing the interface and vice versa.

With NMS, requirements have changed repeatedly in the several years I've been working on the application. As I described in "How I Got It," earlier in this paper, the biggest change came when moving from the original hardware to the new version. But there have been many other changes as well, and almost every time, the separation of the business objects from their visual representation has simplified the process.

Similarly, once the basic version of Sudoku was working, I had ideas for improving the game.

Providing hints

First, I realized that I wanted to offer a "hint" button on the form. In order to give the user a hint (that is, fill in one cell), I needed to have the solution available. So my original strategy of using only the fixed cells as input would no longer work.

Choosing a new input format wasn't difficult. To put the entire solution in a text file, comma-separated rows of data representing the rows of the solution made sense to me.

The only tricky question was how to indicate fixed values. I chose to follow those with an asterisk. So a complete data file for a 9x9 game looks like Listing 31.

Listing 31. The data file format for the revised Sudoku game, with hints available, uses one row for each row on the board. Fixed values are indicated by the asterisk following the value.

```
6*,9,1,3*,7,8,2*,5*,4*
7*,8*,2*,4*,1,5,6,3*,9
5,3,4,6,9,2*,7*,8,1*
2,1,5*,8*,6*,9*,3*,4*,7*
4,7,8*,1,5,3,9*,6,2
3*,6*,9*,7*,2*,4*,8*,1,5
8*,5,7*,9*,4,6,1,2,3
9,2*,6,5,3,1*,4*,7*,8*
1*,4*,3*,2,8,7*,5,9,6*
```

In order to have the solution available, we need to store it in the game, so I added a new property, nSolution, to bizCell to hold the solution value for that cell.

The next step was to add a method to bizGame to read and parse this data. While we could make two passes through the data, one to store the solution values, and one to set the fixed data values using bizGame's existing AddFixedData method, it seemed to make more sense to handle both items as once. The new ReadData method reads in the data file, parses it, and then stores the solution values. Like a number of other methods discussed earlier, the actual storage of the data is handled by calling a method of cbzSetOfGroups, which passes the call down through the hierarchy. Ultimately, bizCell's new SetSolution method (shown in Listing 33) is called to do the actual storage.

Listing 32. The new ReadData method of bizGame reads in a data file, parses it, and stores the solution data, setting up the fixed cells at the same time.

```
* Read in the data for this game.
* Data is in a comma-separated text file with
* one line per row. Fixed values are followed by an asterisk.
* For example, this line:
*
* 3,2,7,1*,4*,9,6,5,2
*
* indicates that the 4th cell of this row contains 1 and the
* fifth cell contains 4 when the game begins.

LPARAMETERS cDataFile

LOCAL cContent, aRows[1], aRowData[1], nRow, nCol, nValue, lFixed

cContent = FILETOSTR(m.cDataFile)

IF ALINES(aRows, m.cContent) <> This.nSize
   * Data is missing
   RETURN .F.
ENDIF
```

```
FOR nRow = 1 TO This.nSize
   * Parse this line
   IF ALINES(aRowData, aRows[m.nRow], ",") <> This.nSize
      * Data is missing
      RETURN .F.
   ENDIF

   FOR nCol = 1 TO This.nSize
      IF RIGHT(aRowData[m.nCol], 1) = "*"
         nValue = VAL(LEFT(aRowData[m.nCol], LEN(aRowData[m.nCol])-1))
         lFixed = .T.
      ELSE
         nValue = VAL(aRowData[m.nCol])
         lFixed = .F.
      ENDIF
      This.oRows.SetSolution(m.nRow, m.nCol, m.nValue, m.lFixed)
   ENDFOR
ENDFOR

RETURN
```

Listing 33. bizCell's new SetSolution method stores the solution value for the cell, and if the cell is fixed, sets it value.

```
LPARAMETERS nValue, lFixed

This.nSolution = m.nValue
IF m.lFixed
   This.SetValue(m.nValue, m.lFixed)
ENDIF

RETURN
```

On the UI side, the main work of providing a hint is handled by a new method of cntBoard, GiveHint. This method (shown in Listing 34) finds a random empty cell and sets its value to its solution value. Note the two-step process for setting the value. First, the business object's (bizCell's) value is set, then the method looks up the corresponding cntCell object and sets its value to the new bizCell value.

Listing 34. cntBoard's GiveHint method locates an empty cell and sets it value to its solution value.

```
* Give the user a hint by filling in an
* empty cell.

LOCAL oBizCell, oCell

* Get a random empty cell
oBizCell = This.oBizGame.GetEmptyCell()

IF NOT ISNULL(m.oBizCell)
   * Set its value to its solution
   oBizCell.SetValue(oBizCell.nSolution)

   oCell = This.GetCellByBizCell(m.oBizCell)
```

```
   oCell.SetValue(oBizCell.nValue)
ENDIF

RETURN
```

Handling the jigsaw variation

Once I had the basic game working with hints (as well as the ability to clear the game and to start a new game), I turned to the question of handling the jigsaw Sudoku variant (shown in Figure 9) where blocks can be any shape (though still must be contiguous). I assumed from the beginning that such variants would require subclassing, but as I tackled this variant, I found that I needed to make some changes to the code for the regular game as well, to avoid repeating code in the subclass.

The first step was to figure out how to specify the blocks. I considered several possibilities, and ultimately decided to add one row to the input file for each block; that row would contain a list of the cells in that block. So, for a 9x9 game, the input file contains 18 rows: the first 9 are the solution in the format described above, while the last 9 lay out the blocks. The format for each block is a comma-separated list of pairs; each pair is in the form "row/column". Listing 35 shows an example.

Listing 35. For jigsaw Sudoku, the input file contains both the solution and the list of cells for each block.

```
1*,9,7,2,3,6*,4,5*,8
2,8,4*,5,9*,7,1,6*,3
3*,7*,5,4,1,8,2*,9,6
4,1,6,7,8,2,5,3,9
5,6,3,9,4*,1*,8*,2,7
8*,2,9,6,5,3,7,4*,1
9,4,8,1*,6,5,3*,7,2
7,5,1,3*,2,9,6,8,4*
6*,3,2*,8*,7,4,9,1,5
1/1,1/2,1/3,2/1,2/2,2/3,3/3,4/3,5/3
1/4,1/5,1/6,2/4,2/5,3/4,3/5,4/4,4/5
1/7,2/7,2/8,3/7,3/8,4/7,4/8,5/7,6/7
1/8,1/9,2/9,3/9,4/9,5/8,5/9,6/8,6/9
2/6,3/6,4/6,5/4,5/5,5/6,6/4,6/5,6/6
3/1,3/2,4/1,4/2,5/1,5/2,6/1,6/2,7/1
6/3,7/2,7/3,8/1,8/2,8/3,9/1,9/2,9/3
7/4,7/5,7/6,8/4,8/5,9/4,9/5,9/6,9/7
7/7,7/8,7/9,8/6,8/7,8/8,8/9,9/8,9/9
```

As soon as I'd settled on a data format, I realized that there was a significant problem with my original design for setting up the game. As originally designed, cells are created and added to the groups before data is read. For standard Sudoku, that was no problem, but for the jigsaw variant, we need the block data before we can add cells to their blocks.

To handle this case, I broke the set-up process down into different parts. I modified bizGame.NewGame (which had been added to support starting a new game through the UI),

to receive the name of the data file and then call three other methods to set up the game, as shown in Listing 36.

Listing 36. The NewGame method of bizGame breaks creating a new game into three steps:

```
LPARAMETERS cGameFile

This.ReadData(m.cGameFile)
This.SetupGame()
This.ParseData()

RETURN
```

I also added a property to bizGame to hold the raw data from the input file. In bizGame, ReadData simply reads in the raw data and sets the game size, as shown in Listing 37.

Listing 37. In bizGame, ReadData just grabs the raw data and determines the game size.

```
LPARAMETERS cDataFile

LOCAL cContent

This.cRawData = FILETOSTR(m.cDataFile)

This.SetGameSize()

RETURN
```

bizGame.SetupGame is unchanged from the version shown in Listing 4. The ParseData method contains the parsing code that was previously in ReadData (Listing 32), though it expects to find the data to parse in the cRawData property rather than reading it from a file.

With those changes made, I could subclass bizGame to create bizGameJigsaw. The main difference is the code in AddCellToBlock, shown in Listing 38, which uses the data from the input file to figure out which block to put the cell into.

Listing 38. In bizGameJigsaw, AddCellToBlock uses the input file data to determine into which block to put a specified cell.

```
LPARAMETERS oCell, nRow, nColumn

* Look up the row, column pair in the data
* to determine which block this cell goes in.

LOCAL cPair, nBlock, nPos, cDataRow, nPosInBlock, aDataRows[1], aBlockData[1]

cPair = TRANSFORM(m.nRow) + "/" + TRANSFORM(m.nColumn)

* Figure out which row of the raw data it's in
nPos = AT(m.cPair, This.cRawData)
IF m.nPos > 0  && this test should never fail
   * The row contains the data for this block
   nBlock = OCCURS(CHR(13), LEFT(This.cRawData, m.nPos)) - This.nSize + 1
```

```
   ALINES(aDataRows, This.cRawData)
   cDataRow = aDataRows[This.nSize + m.nBlock]
   * Now find this pair in the block data to get the position in the block
   ALINES(aBlockData, m.cDataRow, ",")
   nPosInBlock = ASCAN(m.aBlockData, m.cPair)
   This.oBlocks.AddCell(m.oCell, m.nBlock, m.nPosInBlock, "B")
ENDIF

RETURN
```

Jigsaw Sudoku requires just one change on the UI side. I subclassed cntBoard to create cntBoardJigsaw, and overrode the AddDividers method to draw the lines dividing the blocks. That code is shown in Listing 39.

Listing 39. The AddDividers method in cntBoardJigsaw is more complex than the version in cntBoard.

```
* Add the dividers for this game. Strategy is to fill
* an array the shape of the board with the block number
* that each cell belongs to, and then use that data
* to figure out where to add lines.

LOCAL aBlockLayout[This.nSize, This.nSize]
LOCAL oCell, oBlock, nBlock, nRow, nCol
LOCAL nCellHeight, nCellWidth, cLineName

FOR nBlock = 1 TO This.nSize
   oBlock = This.oBizGame.oBlocks.GetGroup(m.nBlock)
   FOR EACH oCell IN m.oBlock FOXOBJECT
      nRow = oCell.nRow
      nCol = oCell.nColumn
      aBlockLayout[m.nRow, m.nCol] = m.nBlock
   ENDFOR
ENDFOR

nCellHeight = This.GetCellHeight()
nCellWidth = This.GetCellWidth()

* Now add vertical lines by going through each row
* and seeing where blocks end.
FOR nRow = 1 TO This.nSize
   nBlock = aBlockLayout[m.nRow, 1]
   FOR nCol = 2 TO This.nSize
      IF aBlockLayout[m.nRow, m.nCol] <> m.nBlock
         * Add a divider
         cLineName = "linVerticalR" + TRANSFORM(m.nRow) + "C" + TRANSFORM(m.nCol)
         This.NewObject(m.cLineName, "linDividerVertical", "Components.VCX")
         oLine = EVALUATE("This." + m.cLineName)

         WITH oLine
            .Left = (m.nCol-1) * (m.nCellWidth + This.nSpaceBetween) + ;
                    This.nSpaceBetween/2
            .Top = (m.nRow-1) * (m.nCellHeight + This.nSpaceBetween) + ;
                   This.nSpaceBetween/2
            .Height = m.nCellHeight + This.nSpaceBetween
```

```
            .Visible = .T.
         ENDWITH

         * Now watch the new value
         nBlock = aBlockLayout[m.nRow, m.nCol]
      ENDIF
   ENDFOR
ENDFOR

* Now add horizontal lines the same way
FOR nCol = 1 TO This.nSize
   nBlock = aBlockLayout[1, m.nCol]
   FOR nRow = 2 TO This.nSize
      IF aBlockLayout[m.nRow, m.nCol] <> m.nBlock
         * Add a divider
         cLineName = "linHorizontalR" + TRANSFORM(m.nRow) + "C" + TRANSFORM(m.nCol)
         This.NewObject(m.cLineName, "linDividerHorizontal", "Components.VCX")
         oLine = EVALUATE("This." + m.cLineName)

         WITH oLine
            .Left = (m.nCol-1) * (m.nCellWidth + This.nSpaceBetween) + ;
                    This.nSpaceBetween/2
            .Top = (m.nRow-1) * (m.nCellHeight + This.nSpaceBetween) + ;
                   This.nSpaceBetween/2
            .Width = m.nCellWidth + This.nSpaceBetween
            .Visible = .T.
         ENDWITH

         * Now watch the new value
         nBlock = aBlockLayout[m.nRow, m.nCol]
      ENDIF
   ENDFOR
ENDFOR

RETURN
```

With these changes, Jigsaw Sudoku works. No changes are needed to any of the other business classes or to any of the other UI classes.

What I've learned

Working on NMS and, to a lesser extent, Sudoku finally made sense of business objects for me. I've been truly amazed at the ease with which I've been able to implement some changes in NMS, both in the interface and the business logic. Often, the hardest part has been understanding the new requirement, and the change itself has required only a few lines of code.

What a robust set of business objects gives you is a clear separation between the underlying rules of your application and the graphical objects you use to represent them to the user. Creating business objects encourages you to keep business logic out of the user interface and user interface code out of the business logic, as well as to put business logic in one place and one place only.

The complete code for the final version of the Sudoku game is included in the conference materials.

Acknowledgments

My thanks to RFL Electronics Inc. for allowing me to show and discuss the NMS application. Whil Hentzen has been an integral part of getting NMS running.