Understanding Business Objects, Part 2

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.

In my last article, I explained what business objects are and why I had a hard time learning to use them. Then, I looked at a client application (called NMS) that brought the ideas home to me. In this issue, I'll begin to look at the details of using business objects.

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 the ideas around business objects concrete, I built a fairly simple Sudoku game. In case you're not familiar with Sudoku, 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 1. The figure also shows the three types of cell groups: rows, columns, and blocks.

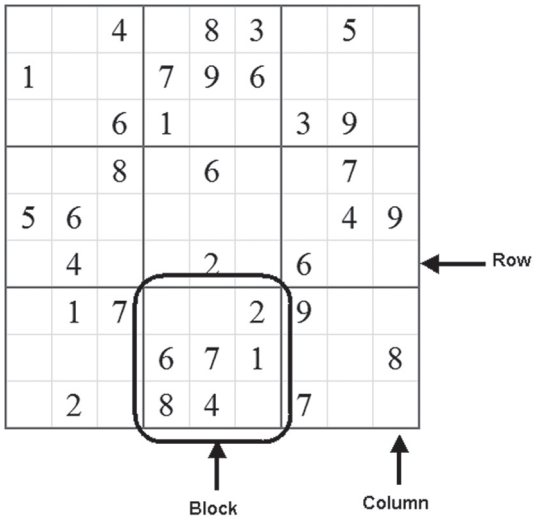

Figure 1. The standard Sudoku game features a 9x9 grid divided up into nine 3x3 squares. (Game captured from http://www.websudoku.com/.)

Initially, some subset of the cells contains numbers between 1 and the grid size (in fact, any set of symbols can be used, but numbers are most common). 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 2 shows the solution for the game in Figure 1. (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 2. The solution to the puzzle in Figure 1. 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 3. 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.

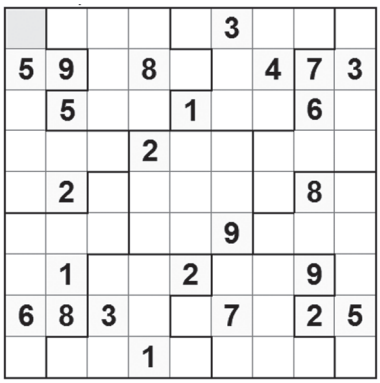

Figure 3. This Sudoku variation retains the overall square shape, but the blocks are not squares. In this version, the rules are still that each number from 1 to 9 must appear once in each row, once in each column and once in each block indicated by the darker lines. (Jigsaw Sudoku captured from http://www.sudokusplashzone.com)

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 (described in some detail in my last article), 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 4 shows (a simplified version of) the hierarchy of objects and collections.

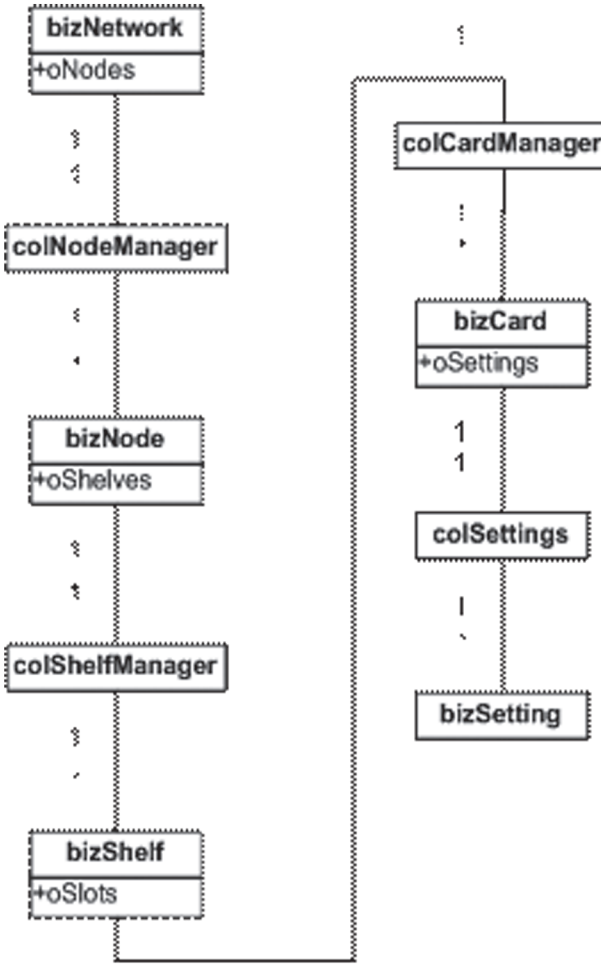

Figure 4. The business object hierarchy for the Network Management System features alternating scalar objects and collections.

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 business objects really show their 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 5 shows the hierarchy of business objects.

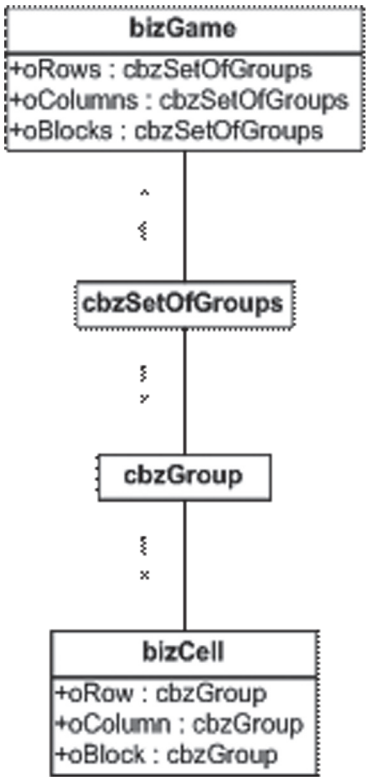

Figure 5. Like the NMS hierarchy, the object hierarchy for Sudoku uses a mix of collections and scalar objects.

Implementing business objects in VFP

As I said in my last article, 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; 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). For the reasons I prefer collections to arrays, see my article in the November, 2009 issue.

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) classes.

My experience has been that collection classes work especially well as "manager" objects. In my client's application, the scalar business objects like bizNode and bizShelf are consolidated into manager collections like colNodeManager and colShelfManager. Although the two collection-based classes in Sudoku aren't named as managers, the role they serve is somewhat managerial.

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 holding 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, that is, object references to other business objects. 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 article.)

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 the CleanUpReferences method of bizGame contains the code in Listing 3.

Listing 3. Cleaning up object references needs 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 five 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 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.

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 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.) Arbitrarily, 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 article.

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 used a comma-separated text file for each game, containing one line for each fixed value. Each line was 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.b 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. Later in this series, I'll describe 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" or "Has." (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, 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. (It is actually possible for one set of groups to be complete, while another is not.)

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 loops through a set of groups, 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 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 is
* 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
* 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.

Engine complete

Once you've built a set of business objects that handle the types of operations described above, you have an engine for your application. At this point, you can perform whatever task the application is for by instantiating your business objects and calling on their methods. For Sudoku, for example, you could play the game from the Command Window by instantiating bizGame, handing it some initial data and then making calls to the SetValue method to change data and to IsComplete and CheckForConflicts to find out whether you've finished the game or introducted conflicts.

However, few users want to use an application that way and certainly, Sudoku wouldn't be much fun without a user interface. In my next article, I'll look at how you connect business objects to the user interface.

The downloads for this article include the complete Suduko game.

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.