Understanding Business Objects, Part 3

Once you have business objects, you need to connect them to the user interface. Plus changing the application is easier than when business logic and UI code are mingled

Tamar E. Granor, Ph.D.

In my last few articles, I introduced the idea of business objects, talked about why I had trouble learning to work with them, and walked through the kinds of code you put in business objects. In the final installment of this series, I'll show how to connect business objects to user interface objects and talk about handling changes in requirements.

My last two articles introduced two applications, NMS and Sudoku. NMS, developed for a client, allows users to monitor and manage hardware in utility substations; the first article in this series contains an overview of this application. Sudoku is an implementation of the popular logic puzzle. The second article in this series explains how the puzzle works, as well as describing the business objects and showing much of their code.

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 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 a 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 1 shows the game board.

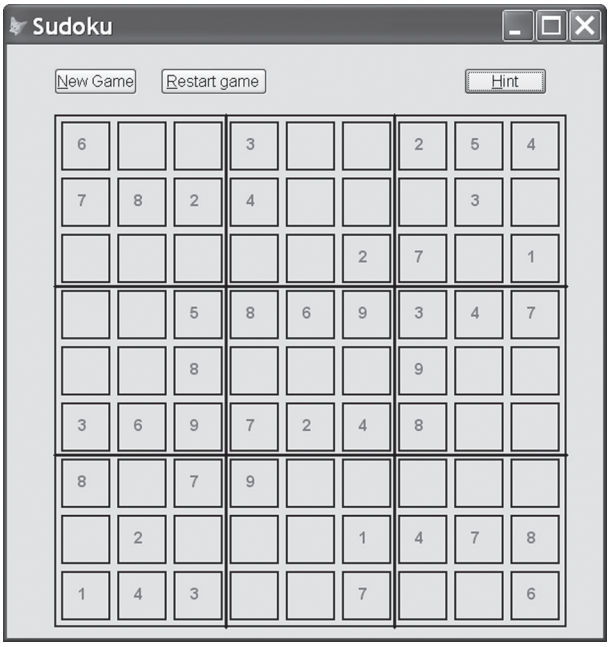

Figure 1. The Sudoku form contains a cntBoard object and a few buttons. Form code handles construction of the board, as well as calling on cntBoard and the business objects to implement the button actions.

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 1) 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 1. 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
```

AddCells, shown in Listing 2, 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 2. 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 3. 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 4. 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 individual ports on the cards 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 5). 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 5. 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 6) 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 7. Once the game is complete, it prevents the user from making further changes.

Listing 6. 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 7. 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 the CheckForConflicts method of cntBoard (Listing 8). 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 8. 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 the first article in this series, 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 9.

Listing 9. 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 (Listing 10) 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 11) is called to do the actual storage.

Listing 10. 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.

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 11. 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 12) 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 12. 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 the second article in this series) 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 13 shows an example.

Listing 13. 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 14.

Listing 14. 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 15.

Listing 15. 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 the second article in this series. The ParseData method contains the parsing code that was previously in ReadData (Listing 10), 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 16, which uses the data from the input file to figure out which block to put the cell into.

Listing 16. 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, nPosIn-
Block, 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 17.

Listing 17. 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 this month's downloads

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, now available. 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.