THE CODEBOOK NEWS

March – April 1998

Volume 2, Issue 2

TABLE OF CONTENTS

EDITOR'S COMMENTS **2**

CODEBOOK REPORT PROCESSING - EXPANDED **3**

Introduction 3
Modifying the Report List Table 3
Adding Pre- and Post-Report Processing to Codebook 4
Hooking Pre- and Post- Processing into the Reporting Engine 4
Creating the PreReportProcessing() Functionality 5
Creating PostReportProcessing() Functionality 6
Using the New Functionality 6
Creating the Report Processing Program 6
The Next Step: Overriding Default Reporting Behavior 7
Modifying the cmdRun.Click() Method ... Again 7
Modifying the REPOLIST.DBF ... Again 8
Override Concepts 8
Print To File Processing 8
Using the New Functionality 9
One Final Enhancement 10
Hooking the Output Destination Radio Button to the Enhancement 11
Conclusion 12

APPLICATION SECURITY **26**

Introduction 26
Defining Interface Objects 27
Security Labels 28
Security Label Naming Conventions *28*
Security Label Grouping *28*
Populating the Interface Objects Table 29
Populating the User Table 29
Activating SAVI Application Security 31
Changing the Security Bypass Keyword 32
Defining User/Interface Object Application Security 32
The Final Step 34
Identifying the User at Login *34*
Two Implementations *34*

EXTENDING COLLECTION CLASSES **36**

Introduction 36
CReferenceCollection 37
Adding Functionality 37
Conclusion 39

SAVI CODEBOOK APPLICATION FORM **41**

Editor's Comments

Charles T. Blankenship

An apology is in order. Due to recent developments, I was unable to submit this issue to the associate editors for review before its publication not to mention this issue is a month late <s>. DevCon and being hired by Flash Creative Management, Inc. created a delay in the publication cycle. As a result, I must rely on all of you to provide editorial input. Therefore, if you find typographical or conceptual errors, let me know via eMail. They will be incorporated into the next version of this issue and republished. You are about to realize just how much the associate editors contribute to this publication, pro bono publico.

The article **Codebook Report Processing – Expanded** illustrates how easy it is to enhance the Codebook framework. In just about all reporting situations I have been presented with the need to perform pre- and post-report processing as well as be able to define exactly how a report should be printed to file, previewed and printed to printer. This article explains how to enhance the Codebook reporting engine to provide this functionality.

The article **Application Security** defines how a developer can activate the SAVI Codebook framework security. It defines the theory and practice behind SAVI's application level security enhancement. If you are using the SAVI version of Codebook, please work through this article when you set up your application security. Special thanks goes to Jun Zhao for the help provided with this article.

The article **Extending Collection Classes** was submitted by Michael G. Emmons, president of Z-Buffer Data Technologies. This article discusses one way of modifying the framework to ensure that a user can instantiate only a specified number of instances of a form.

Codebook Report Processing - Expanded

Codebook reporting, in its native form, meets many *simple* reporting needs. Many times, however, the need to expand its functionality arises. This enhancement provides you with the capability to attach pre- and post-processing to Codebook reporting as well as override the native output processing of previewing, printing to file and printing to printer.

CTBlankenship

Introduction

Adding this functionality to the Codebook reporting engine is a very simple modification. The following steps document everything you must do to accomplish it. The final section of this article illustrates how to implement your new creation. Also, if you find you do not want to make these modifications yourself but would like to have them handed to you on a silver platter, you can always request SAVI's version of the framework, available for free, by filling out the application form at the end of this issue.

Modifying the Report List Table

Codebook's report information is stored in a table, located in the METADATA subdirectory of your application. Its name is REPOLIST.DBF. The original structure of this file follows:

```
cDOSName      Character     8
cFullName     Character     30
cType         Character     4
```

This table stores the following things:

1. The DOS file name for the report sans the file extension, e.g. "MNUMRPT" for the report file "MNUMRPT.FRX"
2. The full name of the report, "MNUM Report" which is what the user sees when they activate the Codebook reporting form
3. The type of report, "**`REPO`**" for Report, "**`LIST`**" for Listing. The value you place in this column determines which report names are viewed when the user selects the Listings or Reports option button for the Output Type on the Print dialog.

This is all you get for reporting from Codebook. It makes the assumption that all processing you need to do can be accomplished using a local or remote view. The result of which is then presented to the report and printed to the printer.

I have found, in actual development, that I need much more than this. Many times I need to hook pre-report processing into the application in order to condition the data properly before the report is printed. I need post-report processing to clean up the environment after the report has printed. I also need to be able to override the native output choice, (Print To File, Print To Printer and Print Preview) This article describes how to accomplish all of this.

Adding Pre- and Post-Report Processing to Codebook

In order to add pre- and post-processing functionality to the reporting engine add two new columns to the REPOLIST table. This table is located on the DATA tab of your project. Look under the Free Tables node and select the REPOLIST table. Press the MODIFY button and add the following fields:

```
cPreProc      Character     70
cPostProc     Character     70
```

The purpose of the cPreProc field is to store the name of the program to execute before the report runs. The purpose of the cPostProc field is to store the name of the program to execute after the report runs.

Hooking Pre- and Post- Processing into the Reporting Engine

The report processing is located in a class definition named cReportForm which, in turn, is located in the cCustFrm.VCX class library. Modify this form and add two methods to it with the following names:

```
PreReportProcessing()
PostReportProcessing()
```

Ensure that the scope for each of these new methods is PUBLIC.

Next, modify the Click method of the cmdRun button and add the following code. The entire listing for this method is located at the end of this article.

```
*-----------------------------------------
*-- MODIFIED Friday, 10/24/97 11:20:53 - CTB:
*-----------------------------------------
llOK2Print = THISFORM.DisplayPrePrintingDialog()

IF llOK2Print
   llPreReportProcessingStatus = THISFORM.PreReportProcessing()
ENDIF

DO CASE
  CASE .NOT. llOK2Print
     WAIT WINDOW "Printing cancelled ... " TIMEOUT 2

  *-------------------------------------------------------------
  *-- If the pre-report processing failed, do not print the report
  *-------------------------------------------------------------
  CASE .NOT. llPreReportProcessingStatus

  CASE thisform.opgOutput.optPreview.Value = 1
    THISFORM.PrintPreview( lcSeleReport )

  CASE thisform.opgOutput.optPrinter.Value = 1
    THISFORM.PrintToPrinter( lcSeleReport )
```

```
  CASE thisform.opgOutput.optFile.Value = 1
    THISFORM.PrintToFile( lcSeleReport )

ENDCASE
```

For now, ignore the **`DisplayPrePrintingDialog()`** method call. This is code, present in the SAVI Version of Codebook, allows a developer to identify the name of a form that should be displayed before the print job is executed. This is convenient when you need to accept filtering information from the user prior to processing the report.

Notice that if the processing performed in either the **`DisplayPrePrintingDialog()`** or **`PreReportProcessing()`** methods fails that the report is not printed. However, if the user accurately provided the needed information and the pre-report processing was successfully executed, the report prints as specified.

Creating the PreReportProcessing() Functionality

Several things must happen in the **`PreReportProcessing()`** method. First, the name of the program to execute before the report is printed must be extracted from the REPOLIST table. Second, the specified program must be executed and its return value received and returned to the cmdRun method. Remember that if the return value is .T. the report will be printed. If the return value is .F. the report will not be printed. Third, this processing must be backwards compatible. In other words, if this class definition is being run against a REPOLIST table that does not have a cPreProc field then this processing must be skipped and program execution must continue as normal.

Place the following code in the **PreReportProcessing()** method:

```
LOCAL lcReportPreProcessing , ;
      llRetVal

llRetVal = .T.

IF TYPE( "RepoList.cPreProc" ) == "C"
   IF .NOT. EMPTY( RepoList.cPreProc )
      lcReportPreProcessing = ALLTRIM( RepoList.cPreProc )
      llRetVal = &lcReportPreProcessing
   ENDIF
ENDIF

RETURN llRetVal
```

Notice that the first step is to default the return value to .T. This is done in order to ensure that the report is printed as it normally would have been if this processing fails for one of the following reasons. The next step takes care of the backwards compatibility issue, it checks if the field repolist.cpreproc exists. If the field exists, it is checked to determine if a pre-processing program was specified. If a pre-processing program is specified, the name of that program is extracted from the table and executed using macro substitution. The return value is captured in the variable llRetVal and returned to the calling program, cmdRun::Click() in this case.

Creating PostReportProcessing() Functionality

The processing for the **PostReportProcessing()** method is much the same as for the **PreReportProcessing()** in that it must take into consideration backwards compatibility and it must execute the specified post report processing program if it exists. Place the following code in the form's **PostReportProcessing()** method:

```
LOCAL lcRepoListFile , ;
      lcReportPostProcessing, ;
      llRetVal

llRetVal = .T.

IF TYPE( "RepoList.cPostProc" ) == "C"
   IF .NOT. EMPTY( RepoList.cPostProc )
      lcReportPostProcessing = ALLTRIM( RepoList.cPostProc )
      llRetVal = &lcReportPostProcessing
   ENDIF
ENDIF

CLOSE TABLES

IF .NOT. USED('repolist')
   lcRepoListFile = ".\metadata\repolist"
   USE ( lcRepoListFile ) IN 0
ENDIF

RETURN llRetVal
```

Using the New Functionality

This design allows you to identify a different program to execute for pre- and post-processing for each report. To specify a pre-processing program for a report, place the name of the program in the **cPreProc** column. For example, if the name of the program is MNUMPRE.PRG place "**MNUMPRE**" in the **cPreProc** column. To specify a post-processing program for a report, place the name of the program in the **cPostProc** field for the report. For example, if the name of the program is MNUMPOST.PRG place "**MNUMPOST**" in the **cPostProc** field. Notice that the .PRG extension for the program file is omitted.

Implementing this new functionality in this method requires that you have a different program for the pre and post processing of each report. This can result in many, many programs performing pre- and post-report processing functionality. A better approach is to create one program whose sole function is to perform report processing and pass it different parameters to identify which report is being processed.

Creating the Report Processing Program

Create a program named REPPROC and add it to the project. Modify this program and add the following code. You can place this program in either the REPORTS or PROGS subdirectory. A good argument can be made for both locations.

```
LPARAMETERS tcProcessing
LOCAL llRetVal = .T.
DO CASE
CASE tcProcessing = ""
CASE tcProcessing = ""
OTHERWISE
ENDCASE

RETURN llRetVal
```

Use the parameter to identify which type of the processing you want to accomplish for each report. For example, to execute the MNUM report's pre-processing, enter the following into the **cPreProc** field, **REPROC('MNUM REPORT PRE-PROCESSING')**. For the MNUM report's post-processing, enter the following into the **cPostProc** field, **REPPROC('MNUM REPORT PRE-PROCESSING')**.

Just for giggles and grins I gave you the REPPROC listing for a working application. See Listing Two to see what this code looks like in a single user, production environment (also note that this code was not written for nor expected to be migrated into a client-server environment ... this makes it much easier and cheaper to write by the way). Also included with this issue's .ZIP file is the corresponding REPPROC.DBF. You should examine both to see how the file and the program relate to one another. Drop me an eMail if you have any questions, mailto:ctb@savvysolutions.com.

The Next Step: Overriding Default Reporting Behavior

The previous discussion illustrated how easy it is to expand Codebook report processing by hooking pre- and post-reporting processing into the Codebook reporting engine. The next step is to provide a way to override the default processing of printing to file, printer or previewing the print job.

This particular enhancement evolved when a client of mine needed to print a report to a file. The problem was that the file had to be a comma delimited text file in a predefined format. It needed to be used as input into a mainframe accounting application ... uh oh.

The default behavior of Codebook is to print the result of the .FRX to a text file ... in the same format as the report. This was unacceptable. Unfortunately, their users still had to be able to use the dialog provided by cReportForm to create the file. In this case, the Print To File functionality had to be overridden and replaced with my own. Fortunately, the same concept used to provide pre- and post-report processing can be used to override the default reporting behavior of Codebook.

Modifying the cmdRun.Click() Method ... Again

If you examine the new code, you'll notice that three new methods were added to the cReportForm class definition.

```
DO CASE
  CASE .NOT. llOK2Print
```

```
    WAIT WINDOW "Printing cancelled ... " TIMEOUT 2

  *-------------------------------------------------------------
  *-- If the pre-report processing failed, do not print the report
  *-------------------------------------------------------------
  CASE .NOT. llPreReportProcessingStatus

  CASE thisform.opgOutput.optPreview.Value = 1
    THISFORM.PrintPreview( lcSeleReport )

  CASE thisform.opgOutput.optPrinter.Value = 1
    THISFORM.PrintToPrinter( lcSeleReport )

  CASE thisform.opgOutput.optFile.Value = 1
    THISFORM.PrintToFile( lcSeleReport )

ENDCASE
```

Each of these new methods encapsulate the complexity of exactly *how* the report is previewed, printed to the printer or printed to a file (the one in which we are interested).

Note: this enhancement to the Codebook reporting engine originated because I needed to override the default behavior of printing to a file. The capability to do the same thing with previewing a report and printing it to the printer resulted simply as a logical extension of the original enhancement requirement. You may or may not find a use for them.

Modifying the REPOLIST.DBF ... Again

A place must be provided for you to define the programs that you want to execute (instead of the default Codebook behavior) for printing to file, printing to printer and previewing a report. This requires yet another modification to the REPOLIST table. Add the following columns:

```
cPTPProc      Character     70
cPTFPRoc      Character     70
cPPProc       Character     70
```

Each of these columns provide a place for you to define exactly how a report is printed to the printer, printed to file or previewed.

Override Concepts

As stated previously, I needed a way to specify *how* a report is printed to file. In addition to this, I also needed to be able to provide a way to inherit default behavior as well. Basically, I wanted to provide you with the same capability that Visual FoxPro provides, the ability to completely override default behavior OR to program some pre-processing and then CALL the default behavior of the superclass.

Print To File Processing

The method code that defines how to print to a file is presented in Listing Three. Notice that this

processing uses the same logic as the pre- and post-report processing enhancement does. It first checks to see if the REPOLIST table contains the required modifications. If it does, then that column is checked to see if a process is defined that specifies how a report is printed to file. If the process exists, it is executed.

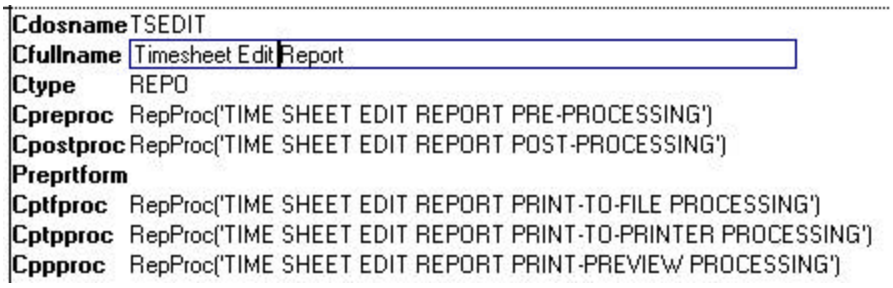
The return value from this process is placed into a variable named **`llDoDefaultBehavior`**. If that variable contains a .T., the original Codebook behavior for printing to file is executed. If it contains a .F., the original code is skipped. This is how I programmatically achieved the OO concept of overriding default behavior.

You must be careful with this design however. Visual FoxPro's default behavior is to return a .T. from a called procedure if a return value is not specifically provided. Therefore, if you do not specifically return a .F. from the Print To File processing that you write, the default Codebook behavior executes as well ... this may or may not provide you with the behavior you expected.

Using the New Functionality

You have the same options available to you as you did in the pre- and post-processing enhancement. You can either specify separate programs to encapsulate your processes or you can use the REPPROC program to encapsulate the entire application's report processing into one location.

The following is a screen capture of the REPOLIST.DBF from a production system. Notice that each enhancement has a process defined in the REPPROC program.



The code in the REPPROC program for the record in the REPPROC table (illustrated above) looks like this:

```
*=====================================================*
*==              TIME SHEET EDIT REPORT             ==*
*=====================================================*
*-----------------------------------------------------
CASE tcProcessing = 'TIME SHEET EDIT REPORT PRE-PROCESSING'
*-----------------------------------------------------
   llRetVal = DoForm( 'frmTimeSheetEditReportPreProcessing' )
*-----------------------------------------------------
CASE tcProcessing = 'TIME SHEET EDIT REPORT POST-PROCESSING'
*-----------------------------------------------------
*-------------------------------------------------------------
```

```
CASE tcProcessing = 'TIME SHEET EDIT REPORT PRINT-TO-FILE PROCESSING'
*-------------------------------------------------------------------
*-----------------------------------------------------------------------
CASE tcProcessing = 'TIME SHEET EDIT REPORT PRINT-TO-PRINTER PROCESSING'
*-----------------------------------------------------------------------
*-------------------------------------------------------------------
CASE tcProcessing = 'TIME SHEET EDIT REPORT PRINT-PREVIEW PROCESSING'
*-------------------------------------------------------------
```

One Final Enhancement

The final request from my client was to be able to specify the default output destination for each report defined for the application. One report may always be printed to a file while another may always be printed to the printer, and yet another may always be previewed before printing. This, thankfully, was a very simple request.

To accomplish this add three more columns to the REPOLIST table as follows:

```
lFile  Logical      1
lpreview     Logical      1
lPrinter     Logical      1
```

If you want the default output destination to be to a file, place a .T. in the .lFile column. If you want the default output destination to be a preview, place a .T. in the .lPreview column. If you want a the default output destination to be to the printer, place a .T. in the .lPrinter column.

Hooking the Output Destination Radio Button to the Enhancement

What we really want to accomplish here is to set the default output destination for the report to the defined value. To make this happen, add a new method to the cReportForm named **SetOutputDestinationDefault()** and place the following code into that new method.

```
LOCAL lnOldWA

lnOldWA = SELECT(0)

SELECT repolist

DO CASE
*--------------------------------------------------
*-- Ensure backwards compatibility by checking to
*-- see if the new fields exist in the REPOLIST table
*--------------------------------------------------
CASE TYPE('repolist.lPrinter') <> "L" OR ;
     TYPE('repolist.lPreview') <> "L" OR ;
     TYPE('repolist.lFile')    <> "L"

CASE EOF('RepoList')
OTHERWISE

   DO CASE
   *--------------------
   CASE repolist.lPrinter
   *--------------------
      THISFORM.opgOutput.optPreview.Value = 0
      THISFORM.opgOutput.optPrinter.Value = 1
      THISFORM.opgOutput.optFile.Value    = 0
   *-----------------
   CASE repolist.lFile
   *-----------------
      THISFORM.opgOutput.optPreview.Value = 0
      THISFORM.opgOutput.optPrinter.Value = 0
      THISFORM.opgOutput.optFile.Value    = 1
   *--------------------
   CASE repolist.lPreview
   *--------------------
      THISFORM.opgOutput.optPreview.Value = 1
```

```
      THISFORM.opgOutput.optPrinter.Value = 0
      THISFORM.opgOutput.optFile.Value    = 0
   ENDCASE
ENDCASE

SELECT (lnOldWA)

RETURN .T.
```

This code must be executed when the form is instantiated as well as when the user selects a new report. Therefore, place the following code in the **`cReportForm ::Init()`** method:

```
THISFORM.SetOutputDestinationDefault()
```

Also place the following code in the **`InteractiveChange()`** method of the lstReport list box

```
THISFORM.SetOutputDestinationDefault()
```

Now, when a user selects a report, the default output destination is selected for them automatically. Just an FYI, during development, I always place a .T. in the .lPreview column to ensure that I don't waste too much paper by printing to the printer (the default behavior).

Conclusion

The default functionality of the Codebook reporting engine is very basic. However, this is not an excuse to criticize the product. Flash gave us the code; therefore, we can do anything we want to with it.

Listing One: The Creportform.cmdRun.Click() Method Code

```
**************************************************
*-- Method:  CReportForm.cmdRun.Click() - CCUSTFRM.VCX
**************************************************

LOCAL lcSeleReport, ;
      llPreReportProcessingStatus

llPreReportProcessingStatus = .T.
*-------------------------------------------------------------------
*-- Extract the name of the report from the RepoList.cDosName field
*-- construct the path and filename of the report.  I am making the
*-- assumption that the VFP list box is the same as the FPD list box
*-- when you specify fields as a source of the list box's information.
*-- When you select an item, the record pointer in the RepoList.DBF
*-- moves to the corresponding record.
*-------------------------------------------------------------------
lcSeleReport = "REPORTS\" + ALLTRIM(repoList.cdosname) + ".FRX"

*-----------------------------------------------
*-- If the file does not exist, message the user
*-- and skip further processing.
```

```
*-----------------------------------------------
IF NOT FILE(lcSeleReport)
  =MESSAGEBOX(REPORTNOTFOUND_LOC, MB_ICONEXCLAMATION)
  RETURN
ENDIF

*-----------------------------------------
*-- MODIFIED Friday, 10/24/97 11:20:53 - CTB:
*-----------------------------------------
llOK2Print = THISFORM.DisplayPrePrintingDialog()

IF llOK2Print
   llPreReportProcessingStatus = THISFORM.PreReportProcessing()
ENDIF

DO CASE
  CASE .NOT. llOK2Print
     WAIT WINDOW "Printing cancelled ... " TIMEOUT 2

  *-------------------------------------------------------------
  *-- If the pre-report processing failed, do not print the report
  *-------------------------------------------------------------
  CASE .NOT. llPreReportProcessingStatus

  CASE thisform.opgOutput.optPreview.Value = 1
    THISFORM.PrintPreview( lcSeleReport )

  CASE thisform.opgOutput.optPrinter.Value = 1
    THISFORM.PrintToPrinter( lcSeleReport )

  CASE thisform.opgOutput.optFile.Value = 1
    THISFORM.PrintToFile( lcSeleReport )

ENDCASE

IF llOK2Print
   THISFORM.PostReportProcessing()
ENDIF
```

Listing Two: The REPPROC listing for a real application

```
#INCLUDE 'INCLUDE\APPINCL.H'

LPARAMETERS tcProcessing
LOCAL llRetVal

llRetVal = .T.

DO CASE
*-----------------------------------------------
CASE tcProcessing = 'MNUM REPORT PRE-PROCESSING'
*-----------------------------------------------
```

```
   LOCAL loMNUMReport

   loMNUMReport = CREATEOBJ( 'WeeklyMNUMReportEnvironment' )

   IF TYPE('loMNUMReport') == "O" AND .NOT. ISNULL( loMNUMReport )
      =CollectMNUMInformation()
      =CreateReportingTableIndexes()
      =CollectCADInformation()
      =CollectDirectChargeInformation()
      =SetReportOrder()
   ENDIF

*-----------------------------------------------
CASE tcProcessing = 'MNUM REPORT POST-PROCESSING'
*-----------------------------------------------

   =CloseAndEraseMNUMReportWorkFile()

*-------------------------------------------------------
CASE tcProcessing = 'MNUM DELTEK REPORT PRE-PROCESSING'
*-------------------------------------------------------
   llRetVal = DoForm( 'frmMNUMDelTekReportPreProcessing' )

*-------------------------------------------------------------
CASE tcProcessing = 'MNUM DELTEK REPORT PRINT-PREVIEW PROCESSING'
*-------------------------------------------------------------
   goApp.oMessageBox.Show( "The MNUM DelTek report only prints TO FILE" )
   llRetVal = .F.

*-----------------------------------------------------------------
CASE tcProcessing = 'MNUM DELTEK REPORT PRINT-TO-PRINTER PROCESSING'
*-----------------------------------------------------------------
   goApp.oMessageBox.Show( "The MNUM DelTek report only prints TO FILE" )
   llRetVal = .F.

*-----------------------------------------------------------------
CASE tcProcessing = 'MNUM DELTEK REPORT PRINT-TO-FILE PROCESSING'
*-----------------------------------------------------------------

    *-------------------------------------------------------------
    *-- Pre-Condition Invariant - the timesheet view is populated
    *--     with the batch of timesheet entries that need to be
    *--     placed in the DelTek file for export to NY.
    *-------------------------------------------------------------
    llRetVal = GetPermissionToCreatePayrollFile()
    llRetVal = llRetVal AND NoBouncedTimeSheetsExist()
    llRetVal = llRetVal AND AllTimesheetsWereChecked()
    llRetVal = llRetVal AND PerformDelTekTimesheetPrePrintProcessing()
    llRetVal = llRetVal AND ExportTimesheetInformation()

    *-----------------------------------------------------
    *-- Prevents the default Codebook file printing processing
```

```
    *-- from creating a file from the .FRX
    *-----------------------------------------------------
    llRetVal = .F.

*-----------------------------------------------------
CASE tcProcessing = 'MNUM DELTEK REPORT POST-PROCESSING'
*-----------------------------------------------------

*=====================================================*
*==             MISSING TIME SHEET REPORT           ==*
*=====================================================*
*-----------------------------------------------------
CASE tcProcessing = 'MISSING TIME SHEET PRE-PROCESSING'
*-----------------------------------------------------
   IF FILE( 'tCurB10E.DBF' )
      ERASE tCurB10E.DBF
   ENDIF

   IF FILE( 'tCurB10E.CDX' )
      ERASE tCurB10E.CDX
   ENDIF

   llRetVal = DoForm( 'frmMissingTimeSheetReportPreProcessing' )
   llRetVal = llRetVal AND GetListOfCurrentEmployees()
   llRetVal = llRetVal AND FindMissingTimesheets()

*-----------------------------------------------------
CASE tcProcessing = 'MISSING TIME SHEET POST-PROCESSING'
*-----------------------------------------------------
   IF USED('tCurB10E')
      USE IN tCurB10E
   ENDIF

   IF FILE( 'tCurB10E.DBF' )
      ERASE tCurB10E.DBF
   ENDIF

   IF FILE( 'tCurB10E.CDX' )
      ERASE tCurB10E.CDX
   ENDIF
*-------------------------------------------------------------
CASE tcProcessing = 'MISSING TIME SHEET PRINT-TO-FILE PROCESSING'
*-------------------------------------------------------------
*---------------------------------------------------------------------
CASE tcProcessing = 'MISSING TIME SHEET PRINT-TO-PRINTER PROCESSING'
*---------------------------------------------------------------------
*-------------------------------------------------------------
CASE tcProcessing = 'MISSING TIME SHEET PRINT-PREVIEW PROCESSING'
*-------------------------------------------------------------

*=====================================================*
*==              TIME SHEET EDIT REPORT             ==*
```

```
*=====================================================*
*-----------------------------------------------------
CASE tcProcessing = 'TIME SHEET EDIT REPORT PRE-PROCESSING'
*-----------------------------------------------------
   llRetVal = DoForm( 'frmTimeSheetEditReportPreProcessing' )
*-----------------------------------------------------
CASE tcProcessing = 'TIME SHEET EDIT REPORT POST-PROCESSING'
*-----------------------------------------------------
*-------------------------------------------------------------------
CASE tcProcessing = 'TIME SHEET EDIT REPORT PRINT-TO-FILE PROCESSING'
*-------------------------------------------------------------------
*--------------------------------------------------------------------
CASE tcProcessing = 'TIME SHEET EDIT REPORT PRINT-TO-PRINTER PROCESSING'
*--------------------------------------------------------------------
*-------------------------------------------------------------------
CASE tcProcessing = 'TIME SHEET EDIT REPORT PRINT-PREVIEW PROCESSING'
*-------------------------------------------------------------------

*=====================================================*
*==         TIME SHEET EDIT SUMMARY REPORT          ==*
*=====================================================*
*-----------------------------------------------------------------
CASE tcProcessing = 'TIME SHEET EDIT SUMMARY REPORT PRE-PROCESSING'
*-----------------------------------------------------------------
   llRetVal = DoForm( 'frmTimeSheetEditSummaryReportPreProcessing' )

*-----------------------------------------------------------------
CASE tcProcessing = 'TIME SHEET EDIT SUMMARY REPORT POST-PROCESSING'
*-----------------------------------------------------------------
*-------------------------------------------------------------------------
CASE tcProcessing = 'TIME SHEET EDIT SUMMARY REPORT PRINT-TO-FILE PROCESSING'
*-------------------------------------------------------------------------
*--------------------------------------------------------------------------
CASE tcProcessing = 'TIME SHEET EDIT SUMMARY REPORT PRINT-TO-PRINTER PROCESSING'
*--------------------------------------------------------------------------
*-------------------------------------------------------------------------
CASE tcProcessing = 'TIME SHEET EDIT SUMMARY REPORT PRINT-PREVIEW PROCESSING'
*-------------------------------------------------------------------------


OTHERWISE

ENDCASE

RETURN llRetVal

*=========================================================*
*==              Begin Reporting Functions              ==*
*=========================================================*

*------------ MNUM Report -------------
FUNCTION CollectMNUMInformation()
```

```
*--------------------------------------

   =CloseAndEraseMNUMReportWorkFile()

   *-------------------------------------------------------
   *-- ALL MNUM reports use the previous week's week ending
   *-- date (which is always a Sunday) to identify the report
   *-------------------------------------------------------
   ldWEDate = DATE() - DOW(DATE(),1) + 1

   SELECT Mnum.cmnum                       , ;
          Mnum.ctitle                      , ;
          Mnum.dstrt_date                  , ;
          Mnum.dfnl_date                   , ;
          Mnum.dtech_rv                    , ;
          Mnum.dcmpl_date                  , ;
          Mnumaloc.claborcat               , ;
          Mnumaloc.claborcat AS cSort_LC   , ;
          Mnumaloc.csection                , ;
          Mnumaloc.cbranch                 , ;
          Mnumaloc.ntot_alloc              , ;
          Mnumaloc.ntot_used               , ;
          Mnumaloc.nhours_lw               , ;
          Mnumaloc.nhours_tw               , ;
          0000000000.00 AS nCADExpend      , ;
          0000000000.00 AS nCAD_TW         , ;
          0000000000.00 AS nCAD_LW         , ;
          0000000000.00 AS nDirectDol      , ;
          "A" AS cSuppress                 , ;
          ldWEDate AS WE_Date                ;
    FROM  Mnum!mnum INNER JOIN Mnum!mnumaloc ;
      ON  Mnum.cid = Mnumaloc.cmnumfk        ;
    WHERE Mnum.lrpt = .F.                    ;
 ORDER BY Mnum.cmnum                       , ;
          cSort_LC                         ;
INTO TABLE tMNUMRpt

ENDFUNC

*------------- MNUM Report --------------
FUNCTION CreateReportingTableIndexes()
*--------------------------------------
   INDEX ON cMNUM TAG cMNUM
   INDEX ON cmnum + csort_lc + cbranch + csection TAG mnumlc
   SET ORDER TO
ENDFUNC

*------------- MNUM Report --------------
FUNCTION SetReportOrder()
*--------------------------------------
   SELECT tMNUMrpt
   SET ORDER TO mnumlc
```

```
ENDFUNC

*------------ MNUM Report -------------
FUNCTION CollectCADInformation()
*-------------------------------------
    LOCAL lnOldWA
    lnOldWA = SELECT(0)

    SELECT v_CADAllocation
    SCAN
       *-------------------------------------------------
       *-- Only provide CAD Billing information for MNUMS
       *-- that are present in the report
       *-------------------------------------------------
       IF SEEK( v_CADAllocation.cMNUM, "tMNUMRpt", "cMNUM" )
          INSERT INTO tMNUMRpt ( cMNUM                      , ;
                                 cBranch                    , ;
                                 nCADExpend                 , ;
                                 nCAD_TW                    , ;
                                 nCAD_LW                    , ;
                                    cSort_LC                  , ;
                                 cSuppress                  ) ;
                         VALUES ( v_CADAllocation.cMNUM     , ;
                                 v_CADAllocation.cBranch    , ;
                                 v_CADAllocation.nCADExpend , ;
                                 v_CADAllocation.nCAD_TW    , ;
                                 v_CADAllocation.nCAD_LW    , ;
                                 "ZA"                       , ;
                                 "B"                          )
      ENDIF
   ENDSCAN

   SELECT (lnOldWA)

ENDFUNC

*------------ MNUM Report -------------
FUNCTION CollectDirectChargeInformation()
*-------------------------------------
   LOCAL lnOldWA
   lnOldWA = SELECT(0)

   SELECT v_DirectChargeSummary

   SCAN
      IF SEEK( v_DirectChargeSummary.cMNUM, "tMNUMRpt", "cMNUM" )
         INSERT INTO tMNUMRpt ( cMNUM                        , ;
                                nDirectDol                   , ;
                                cSuppress                    , ;
                                cSort_LC                     ) ;
                       VALUES ( v_DirectChargeSummary.cMNUM  , ;
                                v_DirectChargeSummary.nTotal , ;
```

```
                                "C"                             , ;
                                "ZB"                              )
      ENDIF
   ENDSCAN

ENDFUNC

*-------------- MNUM Report --------------
FUNCTION CloseAndEraseMNUMReportWorkFile()
*-----------------------------------------

   IF USED('tMNUMRpt')
      USE IN tMNUMRpt
   ENDIF

   IF FILE('tMNUMRpt.DBF')
      ERASE tMNUMRpt.DBF
   ENDIF

   IF FILE('tMNUMRpt.CDX')
      ERASE tMNUMRpt.CDX
   ENDIF

ENDFUNC

*============================================================*
*==    Begin DELTEK Timesheet Export Report Functions    ==*
*============================================================*

*-----------------------------------------
FUNCTION GetPermissionToCreatePayrollFile()
*-----------------------------------------
    LOCAL lnAnswer, ;
          llRetVal

    lnAnswer = goApp.oMessageBox.Show( "Do you want to create the Payroll File?",
"YesNo" )
    llRetVal = ( lnAnswer = IDYES )

    RETURN llRetVal

ENDFUNC

*---------------------------------
FUNCTION AllTimesheetsWereChecked()
*---------------------------------
   LOCAL lnOldWA, ;
         lnNotChecked, ;
         llRetVal

   lnOldWA = SELECT(0)
```

```
   SELECT v_DelTekTimeSheetExport

   *-----------------------------------------------------
   *-- Determine if ALL timesheets in the batch were checked
   *-----------------------------------------------------
   GOTO TOP

   COUNT ALL FOR v_DelTekTimeSheetExport.lChecked = .F. TO lnNotChecked
   llRetVal = ( lnNotChecked = 0 )

   IF .NOT. llRetVal
      goApp.oMessageBox.Show( "Not all timesheets were checked ... please run
check timesheets again.","Stop" )
   ENDIF

   RETURN llRetVal

ENDFUNC

*---------------------------------
FUNCTION NoBouncedTimeSheetsExist()
*---------------------------------
   LOCAL lnOldWA   , ;
         lnBounced , ;
         llRetVal

   lnOldWA = SELECT(0)

   SELECT v_DelTekTimeSheetExport

   *-----------------------------------------------
   *-- Determine if there are any bounced timesheets
   *-----------------------------------------------
   GOTO TOP
   COUNT ALL FOR v_DelTekTimeSheetExport.lBounced = .T. TO lnBounced
   llRetVal = ( lnBounced = 0 )

   IF .NOT. llRetVal
      goApp.oMessageBox.Show( "Some timesheets are still in a 'bounced' condition
... please correct errors and run check timesheets again.","Stop" )
   ENDIF

   RETURN llRetVal

ENDFUNC

*-----------------------------------------------
FUNCTION PerformDelTekTimesheetPrePrintProcessing()
*-----------------------------------------------
   SELECT v_DelTekTimeSheetExport

   SCAN
```

```
      =TranslateTheMNUMIntoAnAccountCode()
      =TranslateNNSYLaborCategories()
   ENDSCAN
ENDFUNC

*-----------------------------------
FUNCTION ExportTimesheetInformation()
*-----------------------------------
   SELECT dweekend              , ;
          cemp_num              , ;
          filler1               , ;
          paychktype            , ;
          filler2               , ;
          cacctno               , ;
          subacct               , ;
          suffixacct            , ;
          filler3               , ;
          csection              , ;
          filler4               , ;
          cpaytype              , ;
          filler5               , ;
          billhours             , ;
          clabcat               , ;
          filler6               , ;
          tsadjwedat              ;
     FROM v_DelTekTimeSheetExport ;
     INTO CURSOR t_DelTek

     IF _TALLY > 0
        loExportObject = CREATEOBJ('expDelTekTimesheetExport')
        loExportObject.Export()
        loExportObject.Release()
     ENDIF

ENDFUNC

*-----------------------------------------
FUNCTION TranslateTheMNUMIntoAnAccountCode()
*-----------------------------------------
   LOCAL lcMNUM    , ;
         lcAcctNo  , ;
         lcSubAcct , ;
         lcSuffixAcct

   lcMNUM = v_DelTekTimeSheetExport.cMNUM

   DO CASE
   CASE lcMNUM = "ADMIN"
      lcAcctNo     = '0512'
      lcSubAcct    = '010'
      lcSuffixAcct = '00'
   CASE lcMNUM = 'HOLID'
```

```
      lcAcctNo     = '0516'
      lcSubAcct    = '010'
      lcSuffixAcct = '00'
   CASE lcMNUM = 'SICK'
      lcAcctNo     = '0518'
      lcSubAcct    = '010'
      lcSuffixAcct = '00'
   CASE lcMNUM = 'VACAT'
      lcAcctNo     = '0442'
      lcSubAcct    = '000'
      lcSuffixAcct = '00'
   CASE lcMNUM = 'LWOP'
      lcAcctNo     = '0144'
      lcSubAcct    = '000'
      lcSuffixAcct = '00'
   OTHERWISE
      lcAcctNo  = ALLTRIM( SUBSTR( lcMNUM , 1, AT('.', lcMNUM ) - 1))
      lcSubAcct = ALLTRIM( SUBSTR( lcMNUM , AT('.', lcMNUM ) + 1, 3))
      IF LEFT( v_DelTekTimeSheetExport.cEmp_Num, 1 ) = "9"
         lcSuffixAcct = '52'
      ELSE
         lcSuffixAcct = '39'
      ENDIF
   ENDCASE

   REPLACE v_DelTekTimeSheetExport.cAcctNo    WITH lcAcctNo
   REPLACE v_DelTekTimeSheetExport.SubAcct    WITH lcSubAcct
   REPLACE v_DelTekTimeSheetExport.SuffixAcct WITH lcSuffixAcct

ENDFUNC

*-------------------------------------
FUNCTION TranslateNNSYLaborCategories()
*-------------------------------------
   LOCAL lcLabCat

   IF v_DelTekTimeSheetExport.cCustCode = "NNSY"
      DO CASE
         CASE v_DelTekTimeSheetExport.cLabCat = "AA"
            lcLabCat = "01"

         CASE v_DelTekTimeSheetExport.cLabCat = "AB"
            lcLabCat = "02"

         CASE v_DelTekTimeSheetExport.cLabCat = "AC"
            lcLabCat = "03"

         CASE v_DelTekTimeSheetExport.cLabCat = "AD"
            lcLabCat = "04"

         CASE v_DelTekTimeSheetExport.cLabCat = "AE"
            lcLabCat = "05"
```

```
         CASE v_DelTekTimeSheetExport.cLabCat = "AF"
            lcLabCat = "06"

         CASE v_DelTekTimeSheetExport.cLabCat = "AG"
            lcLabCat = "07"

         CASE v_DelTekTimeSheetExport.cLabCat = "AH"
            lcLabCat = "08"

         CASE v_DelTekTimeSheetExport.cLabCat = "AJ"
            lcLabCat = "09"

         CASE v_DelTekTimeSheetExport.cLabCat = "AK"
            lcLabCat = "10"

         CASE v_DelTekTimeSheetExport.cLabCat = "AL"
            lcLabCat = "11"

         OTHERWISE
            lcLabCat = ""

         ENDCASE
   ELSE
      lcLabCat = ""
   ENDIF

   REPLACE v_DelTekTimeSheetExport.cLabCat WITH lcLabCat

ENDFUNC

*-----------------------------------
FUNCTION GetListOfCurrentEmployees()
*-----------------------------------
   LOCAL llRetval

   llRetVal = .NOT. FILE( 'tCurB10E.DBF' )

   IF llRetVal
       SELECT LEFT(emp_num,6) AS cEmp_Num , ;
              ALLTRIM( l_name ) + ", " + ALLTRIM( f_name ) AS cEmp_Name, ;
              .F. AS lMissingTS                    ;
         FROM F:\MANGT\MRSOMS\EMPLOYEE\EMPLOYEE  , ;
              F:\MANGT\MRSOMS\EMPLOYEE\BRANCH      ;
        WHERE employee.cBranchFK = branch.cid AND  ;
              branch.cBrCode = "10" AND            ;
              EMPTY( employee.term_date )          ;
     ORDER BY 2 ;
         INTO TABLE tCurB10E
      *-------------------------------------
      *-- Table name stands for:
      *-- Current Branch 10 Employees tCurB10E
```

```
      *-------------------------------------
      llRetVal = ( _TALLY > 0 )

      IF USED('employee')
         USE IN employee
      ENDIF

      IF USED('branch')
         USE IN branch
      ENDIF
   ENDIF

   RETURN llRetVal
ENDFUNC

*------------------------------
FUNCTION FindMissingTimesheets()
*------------------------------
   LOCAL llRetVal
   llRetVal = .F.
   SELECT tCurB10E

   SET RELATION TO cEmp_Num INTO v_EmployeesWithTimeSheets

   SCAN FOR EOF('v_EmployeesWithTimeSheets')
      *-------------------------------------------------------------
      *-- If a current branch 10 employee does not have a timesheet
      *-- entered, replace lMissingTS with .T., these records
      *-- then appear on the Missing Timesheet Report
      *-------------------------------------------------------------
      REPLACE lMissingTS WITH .T.
      llRetVal = .T.
   ENDSCAN

   SET FILTER TO lMissingTS = .T.

   IF .NOT. llRetVal
      goApp.oMessageBox.Show( "There are no missing timesheets." )
   ENDIF

   RETURN llRetVal

ENDFUNC
```

Listing Three: Printing To File Processing

```
LPARAMETER tcSeleReport

LOCAL lcPrintToFileProcessing , ;
      lcTextFile, ;
      llDoDefaultBehavior, ;
```

```
      llRetVal, ;
      llGotReportFileName

llRetVal = .T.
llDoDefaultBehavior = .T.

IF TYPE( "RepoList.cPTFProc" ) == "C"
   IF .NOT. EMPTY( RepoList.cPTFProc )
      lcPrintToFileProcessing = ALLTRIM( RepoList.cPTFProc )
      llDoDefaultBehavior = &lcPrintToFileProcessing
   ENDIF
ENDIF

IF llDoDefaultBehavior
   *-----------------------------------------------------------
   *-- MODIFIED - 10/25/1997 - CTB:
   *-- Create a DOS text file from the name of the report file
   *-----------------------------------------------------------
   lcTextFile = ""
   llGotReportTextFileName = THISFORM.GetReportTextFileName( @lcTextFile )

   IF llGotReportTextFileName
      *------------------------------
      *-- Print the report to the file
      *------------------------------
      REPORT FORM (tcSeleReport) TO FILE (lcTextFile) ASCII
      *-------------------------------------------------
      *-- Inform the user what the name of the file is.
      *-------------------------------------------------
      =MESSAGEBOX(FILESAVEDAS_LOC + FULLPATH(lcTextFile), MB_ICONINFORMATION)
   ENDIF
ENDIF
```

Application Security

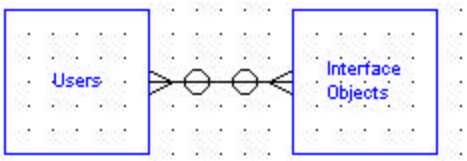
I received many requests to document the use of the security module provided with the SAVI version of Flash Creative Management's Codebook. This article describes how to set up and use this enhancemnt.

C. T. Blankenship

Before I describe anything, it is important to realize that the application security provided by SAVI's version of Codebook is at the interface level, not the data level. It is also important to understand that the data, which is stored in FoxPro files on the LAN or HDD, is just as accessible as it always was. The benefit of this type of security is that it allows an administrator to restrict the activities of unknowledgeable users. If you have users who are fully capable of opening your Visual FoxPro tables and you still want to protect that data, you must couple this interface level application security with a *real* DBMS that secures the data as well. For many companies, this interface level type of security is economical since the people using the application don't have a clue where the data is and second, wouldn't know what to do with it if they did.

Introduction

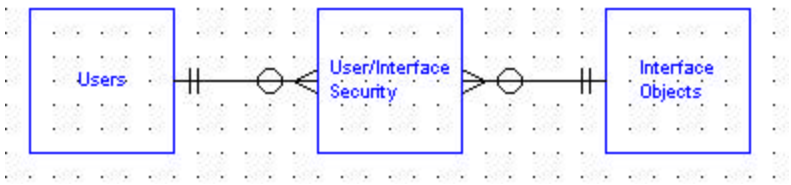
The SAVI security module (located in cSECURE.VCX) is based upon a very simple relationship between two strong entities, **users** and **interface controls**. During the life of the application, each **interface control** can be used by many **users**. Conversely, each **user** is presented with many **interface controls**. The diagram for this relationship is as follows …



Unfortunately, many to many relationships are difficult to implement. They must be simplified by creating an association between the two strong entities. In this case, associating one user with one interface object is the essence of application security. By setting the .Enabled and .Visible properties for each InterfaceObject that is associated with a user, the control can be made invisible, read only or editable. The following table identifies the settings required for the .Enabled and .Visible property to achieve each security level

SAVI Security Matrix	**Enabled Property**	**Visible Property**
Read Only	.F.	.T.
Invisible	N/A	.F.
Editable	.T.	.T.

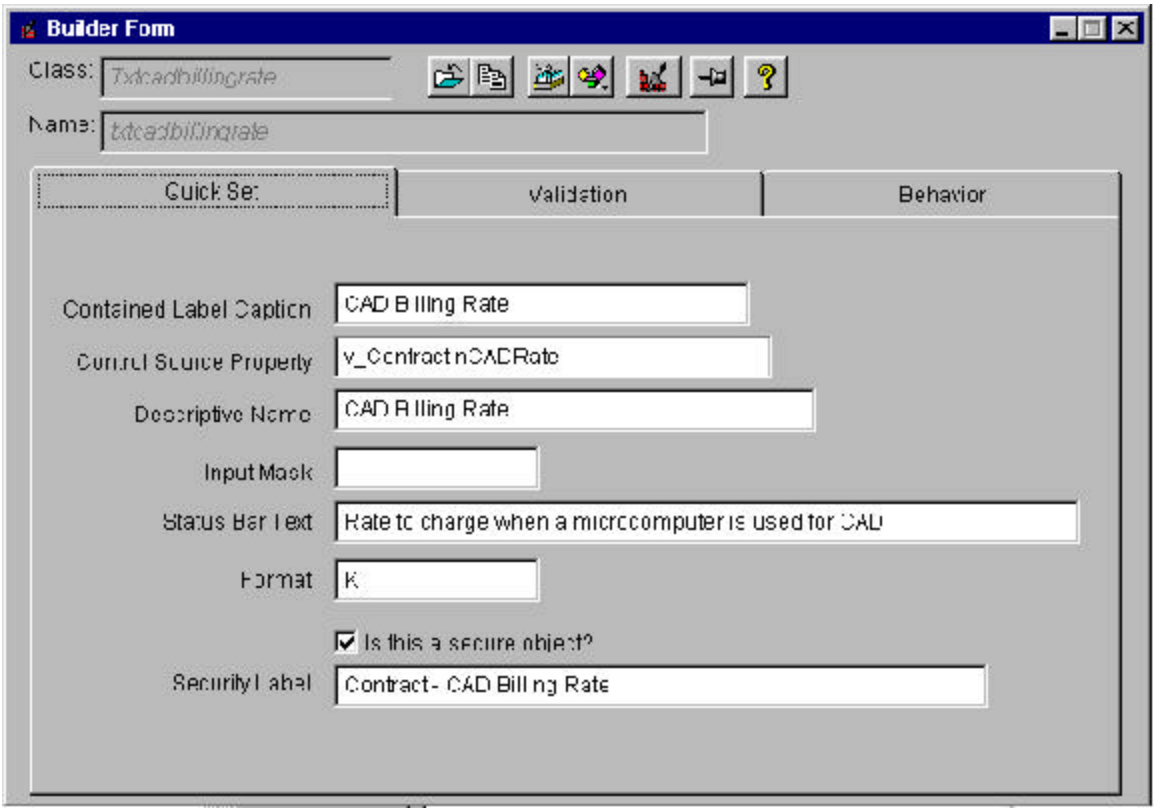
The diagram that illustrates this relationship looks like this:



Once each user is associated with each interface object, the proper setting for the .Enabled and .Visible property of each control can be stored for each user. This is the operational core of the SAVI Application Security module, cSecure.VCX.

Defining Interface Objects

The first step to take when setting up application security is to identify each interface object to the security module. You accomplish this during the application development process by assigning labels to each interface object through the cSecurityLabel property of the SAVI Contained Controls. Each SAVI contained control has a builder associated with it. Each of these builders provides a place for you to identify two things, 1) whether or not the interface object is a secure object and 2) what its security label is. A screen shot of the builder for a SAVI Contained Text Box is as follows:



In this particular situation, the **`txtCADBillingRate`** interface object is modeling the amount of money that one of my clients bills for each hour an employee spends working on a CAD

drawing. As you can probably surmise, it is advisable to allow only a very few of the users (contract managers) to change this value. It is also necessary to allow others to view it but not change it. Finally, it is desirable to prevent most of the users from seeing this confidential information at all. By giving this interface object a descriptive label, **`Contract – CAD Billing Rate`**, this interface object can be associated with any user. Then, one of the three levels of security can be granted to the user by the application administrator.

Security Labels

Security labels present an application developer with a two-edged sword. At one extreme, each interface object can receive a different security label. The other extreme is to give all interface objects the same security label. The correct setting is somewhere between these two extremes. Before we can talk about security labels we first must learn a logical method for naming them.

Security Label Naming Conventions

The rule that I follow when creating security labels is to use the name of the table and the name of the field and separate the two with a hyphen. For example, the CAD Billing Rate column is a member of the Contract table, that is why its security label is **`Contract – CAD Billing Rate`**. Following this logic, you would have a different security label for every control in your application. Doing this will give the application administrator a high degree of control over the application's security. It will also make setting up application security for each user more difficult. There is a happy median that can be reached by using a concept known as security label grouping.

Security Label Grouping

Security label grouping means that more than one control receives the exact same security label. When an application administrator sets security for a control, they are really setting security for a set of logically grouped controls. An excellent example of this is a customer address. At one extreme, you could give each control that defines an address the following security labels:

```
Control Name               Possible Security Label

txtCustomerAddressLineOne  Customer – Address Line One
txtCustomerAddressLineTwo  Customer – Address Line Two
txtCustomerCity                   Customer – City
txtCustomerState                  Customer – State
txtCustomerZipCode                Customer – Zip Code
```

If you did this, the application administrator could assign one user the right to change only the zip code of a customer and revoke their rights to change or even see the other controls. If this is the kind of control your clients need then this is the correct choice.

More than likely, if a user is given the rights to change a customer address they should be given the rights to change all aspects of the customer address. If you give each control the following security labels

```
Control Name               Possible Security Label

txtCustomerAddressLineOne  Customer – Address
txtCustomerAddressLineTwo  Customer – Address
```

```
txtCustomerCity                 Customer – Address
txtCustomerState                Customer – Address
txtCustomerZipCode              Customer – Address
```

the application administrator will be able to grant access rights to the customer address with one mouse click instead of five. If you imagine the impact this can make over an entire, large application, you can begin to see the amount of time that can be saved by the administrator with the intelligent use of Security Label Grouping.

Populating the Interface Objects Table

The first step in activating SAVI security is to give each interface object (that is a secure object) a security label. The second step is to extract those security labels out of the class definitions and load them into the Interface Objects Table.

Security label extraction is performed by the developer using the OBJSEC.PRG located in the SECURITY subdirectory of the application.

To Extract Security Labels

1) Ensure you are in the application's development directory
2) Run the program STARTCB
3) Type DO SECURITY\OBJSEC and press the Enter key

This program first asks for the project file from which you are extracting security labels. Once provided, it opens the project files and identifies each class library that is part of the application. It then opens each class definition file (.VCX) and finds each object that has a property .cSecurityLabel. It extracts the value of this property from the .VCX and tries to add that new security label to TABLE2 in the security system. If the label already exists, it does not add it again. If the label does not already exist it is added.

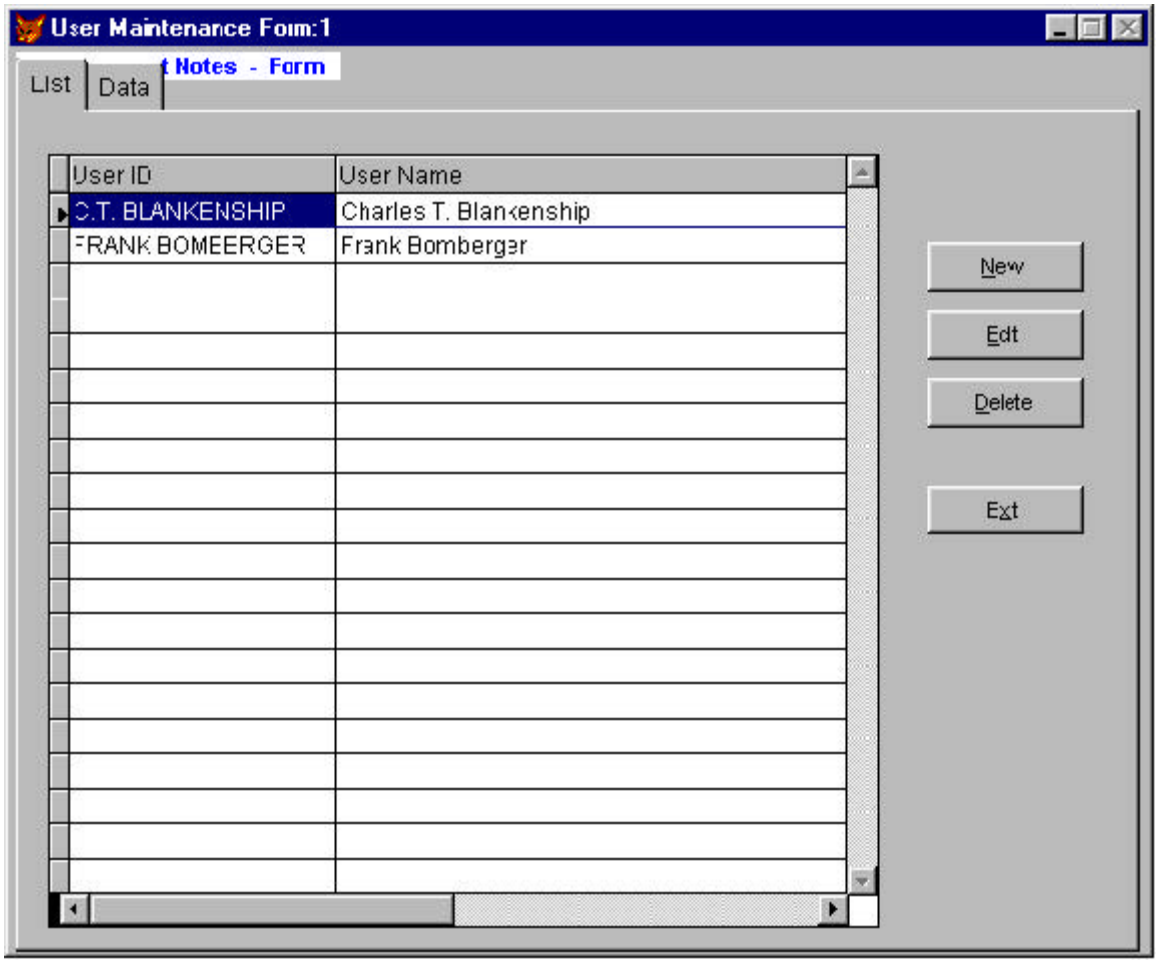
A word of advice. Before you create all of your security labels, you should hold a conference with the employees who are going to be the application administrators. You should get their approval for the security label names as well as the groupings.

Populating the User Table

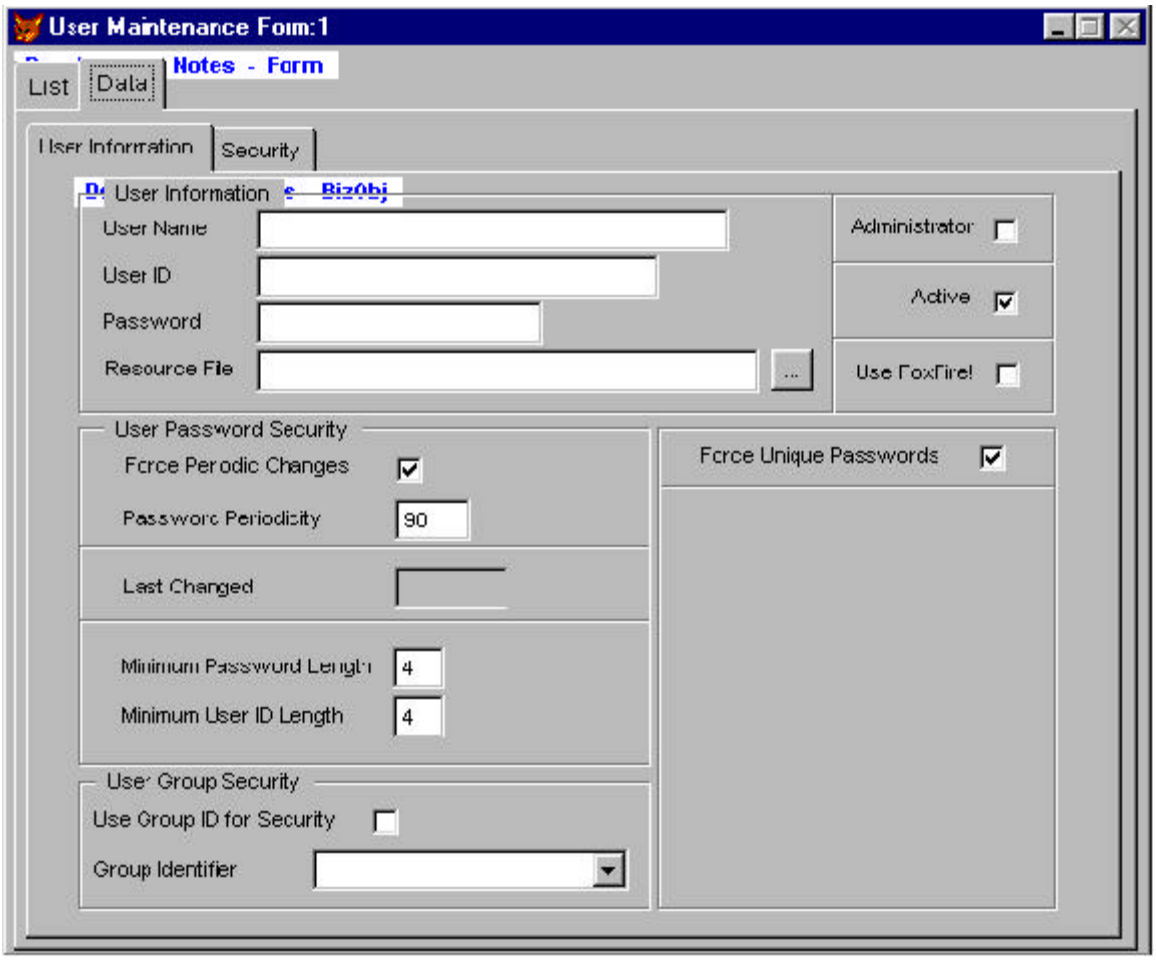
Each new application that you create using QSTART.APP is created in a non-secure mode. This means that a User ID and password is not necessary to gain entry to the application. This allows you to define the first user of the application.

To Populate The User Table

4) Ensure you are in the application's development directory
5) Run the program STARTCB
6) Run the application by typing DO MAIN
7) When the application menu displays, select the Security menu pad
8) From the Security popup, select the Users bar … the following User Maintenance Form displays. Note that your application will not have any users defined at first. This screen shot is from an application that has two users defined, Frank Bomberger and CT Blankenship.



9) Press the New button to add your first user. The following screen displays.



10) Enter the User's Name, this is the given name of the person.
11) Enter the User's ID, this is a code used to identify the person to the application
12) Give the user a password.
13) Specify whether or not the user will be an Administrator (this grants them access to the Security manu pad when the application is on line).
14) Ensure the Active checkbox is checked.
15) Determine whether or not the user is granted access to the FoxFire report writer, checked, or the standard Codebook report writer, unchecked.
16) Determine if you want the user to periodically change the password and what the periodicity is.
17) Define the users minimum Password and User ID length.
18) Determine if you want to assign security to the user based upon the profile of an already defined group.
19) Press the save button.

This process defined your first user and is the method your application administrators will use to populate the User Table.

Activating SAVI Application Security

To secure your application, you must modify a record in the WALSH64.TXT table located in the SECURITY subdirectory.

To Open the WALSH64.TXT Table

20) Ensure you are in the application's development directory
21) Type STARTCB and press the Enter key
22) Type USE WALSH64.TXT and press Enter key
23) Type BROWSE to browse the table. The following window will appear.
24) Change this text string to be anything but **`DynamiteStone`**. For example, **`DynamiteStoneBall`** would work nicely.

DynamiteStone is the default keyword that defeats security in all SAVI applications and allows anyone to log into the application. Passwords and User ID's are no longer required.

Changing the Security Bypass Keyword

It is very advisable to change this password before you distribute your application. Otherwise, anyone who is using the SAVI Codebook framework will know how to gain entry into your application.

To Change the Security Bypass Keyword

25) Ensure you are in the application's development directory
26) Modify the project
27) Select the Programs Code tab
28) Expand the Programs tree node
29) Select and modify SETUP.PRG
30) Near the top of the program you will see the following defined constant

```
#DEFINE  SECURITY_OVERRIDE_STRING       "DynamiteStone"
```

31) Change this string to be anything you want.
32) Change the entry in the WALSH64.TXT table to the same value to defeat security.

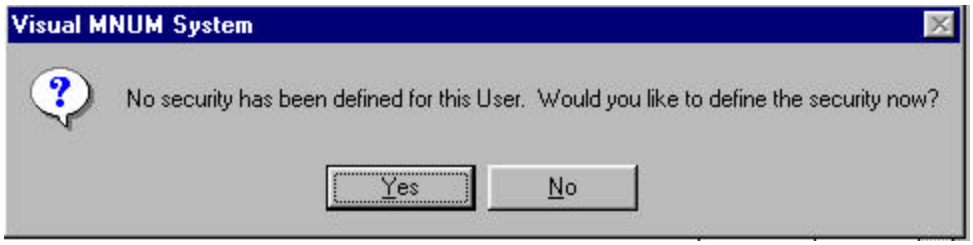
Defining User/Interface Object Application Security

Now that your Interface Objects and at least one User are defined it is easy to associate all interface objects (security labels) with a user.

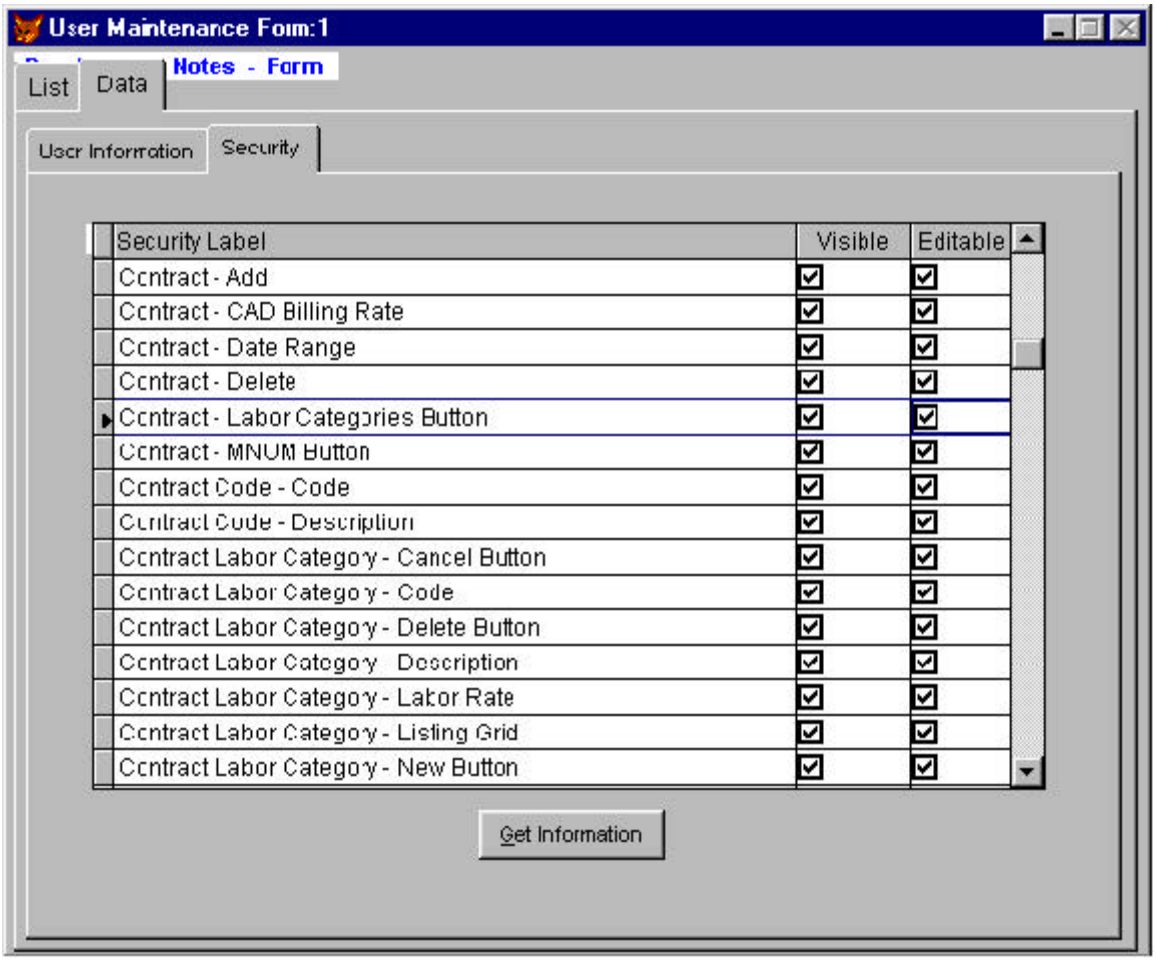
To Associate Security Labels with a User

33) Ensure you are in the application's development directory

34) Type DO STARTCB and press the Enter key
35) Launch the application by typing DO MAIN and press the Enter key
36) Select the Users bar from the Security popup
37) Select the user for which you want to setup application security and press the Edit button
38) When the User Information tab displays, select the Security tab
39) Press the Get Information button. If this is the first time you have assigned security to this user, the following dialog displays:



40) If you select yes, the application creates one record in the User/Interface Security table for each defined Security Label. Once all user / interface object associations are made, you are presented with the following grid.



The process of setting user security is now as easy as checking whether or not you want this particular user to be able to view and/or edit that interface object that has this particular security label.

Notice that the security label I associated with the CAD Billing Rate appears exactly as it was entered into the text box builder several pages back. This is why it is very important to define your interface object security label naming conventions as well as your concepts of using security label grouping. The application administrator is using this information directly to setup application security. It is very important to be clear and precise when communicating with them in this fashion.

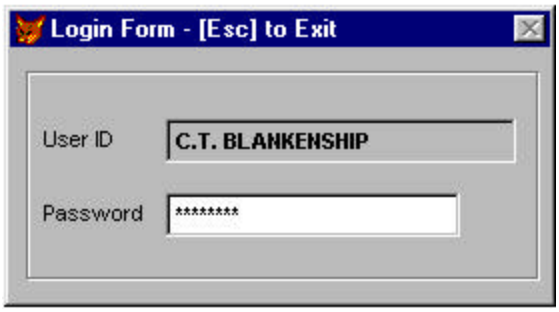
The Final Step

Identifying the User at Login

The final step in setting up security is to create an environmental variable called NET NAME in the CONFIG.SYS file. The line in my CONFIG.SYS looks like this:

```
SET NET NAME=C.T. BLANKENSHIP
```

Notice that the value entered for the NET NAME environmental variable is the EXACT same as the information entered for the User ID when you defined your user. Now, when the users of your application launch your application, they will be presented with a password dialog like the following:



Notice that the user is not able to enter their User ID, it has been pulled out of the environment, displayed and the control disabled. The only piece of information left to enter is a password. This design prevents malicious users from discovering someone else's User ID and purposefully disabling their user account by entering the password incorrectly too many times.

Two Implementations

There are two ways of setting up this environmental variable depending upon how restrictive you want to be with your users.

User/Machine Security

If you want to force your users to only use this application from a specified machine, then place

the environmental variable in that machine's CONFIG.SYS. This can be useful if that particular user does quite a bit of confidential information processing (like payroll) and management only wants that processing to occur in one room of the building.

User/Network Security

If you want the user's to be able to log into the network from any LAN terminal, you can assign this environmental variable to their Network Login using their login script. In this case, their Net Name is still displayed in the Login Form and they are only able to login using their own login. Not someone else's.

Extending Collection Classes

Michael G. Emmons

Introduction

One of the most overlooked areas in the wonderful Codebook framework is its use of collection classes. Put simply, a collection class is an object that contains other objects or references to other objects. Codebook's implementation of the collection object is found in ccollect.vcx. Figure 1 shows the class hierarchy for this class. Much of Codebook's functionality is based upon collections of objects.

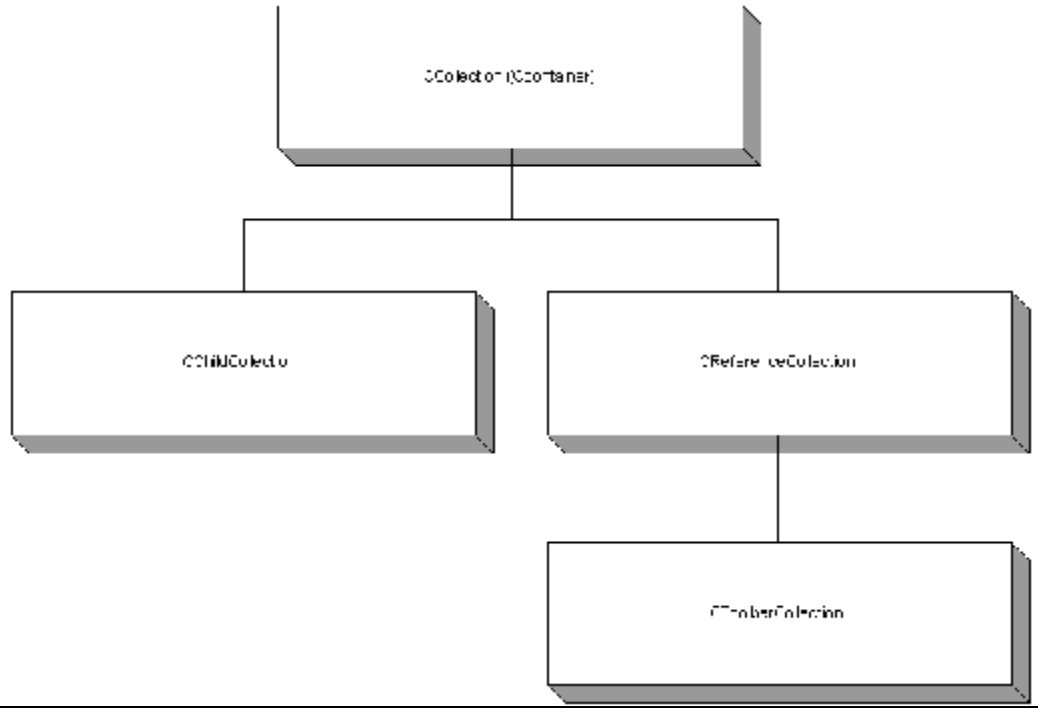


Figure 1

As you can see, there are three basic types of collections used in Codebook: CChildCollection, CReferenceCollection and CToolbarCollection. CChildCollection adds an object as a child of the collection, CReferenceCollection adds an object as a reference and CToolbarCollection allows only one instance of any object to be allowed. For our purposes we will be focusing on CReferenceCollection.

CReferenceCollection

One of the most important functions CReferenceCollection is used for is keeping track of forms. Each time a form is created it adds itself to the application object's forms collection (goapp.oForms). This is used by Codebook for things like staggering multiple instances of the same form and adding instance numbering to the form's title.

With just a little bit of work we can add some pretty neat functionality to forms. For example, a much requested client specification is to be able to limit the number of forms of the same type a user is allowed to open at the same time. This modification could be a major headache for some, but Codebook is so well designed that this is a minor change that should take less than 10 minutes.

What we need is a method to which we can pass a class name and which will return the number of classes of that type currently instanced. If we look carefully at the CReferenceCollection class we see there is already a method that does almost exactly what we want: Scan(). Scan's description says, "Scans the collection for an object or object name passed as a parameter. Returns the index of instance if found, 0 if not." Now, all we have to do is modify the code so that it returns the number of instances found rather than the first instance found and we'll be halfway there.

Adding Functionality

Create a new method called GetInstanceCount in the cReferencecCollection class definition, copy the code from the Scan() method and paste it into the newly created GetInstanceCount() method. Now, instead of returning when the correct instance is found we need to keep a count. So, we need to add a variable called lnFound and set its initial value to 0. Next, look at this bit of code:

```
IF llFound
    RETURN lnRow
ENDIF
```

We need to change this bit of code so that instead of returning as soon as we find a matching class it will increase lnFound. This code should now look like:

```
IF llFound
    lnFound = lnFound + 1
ENDIF
```

Our final change is in the last line: Return 0. Change this to: Return lnCount and we're set! We now have a method that will accept a class or even a name as a parameter and return the number of objects instanced. Below is the complete code:

```
LPARAMETERS tuParm1, tcProperty, tnDirection
*-- tuParm1 can be any property of an object, the actual object, or
*-- a number which represents the row number of this.aInstances[]

*-- tcProperty can be any property of the object contained in
*-- this.aInstances[]. (case insensitive) If not  specified, both
*-- the "Class" and "Name" properties are used in the search.
*-- tnDirection specifies if we want the search to start from the
```

```
*-- beginning (1), or the end (-1) of the collection.

*-- This code was modified from Scan() to return the number
*-- of instances of a name/class found.
IF THIS.IsEmpty()
	RETURN 0
ENDIF

IF TYPE("tcProperty") <> "C"
*-- If we don't specify a property and tuParm1 is not an object,
*-- we'll be searching both the "Class" and "Name" properties.
*-- Therefore, tuParm1 must be of type character.
	IF !INLIST(TYPE("tuParm1"), "C", "O")
		RETURN 0
	ENDIF
	tcProperty = ""
ELSE
	tcProperty = UPPER(tcProperty)
ENDIF

LOCAL llCompareProperty, ;
	lnRow, ;
	lnStart, ;
	lnEnd, ;
	llFound, ;
	lnFound, ;
	lcCompareExpr

llCompareProperty = (TYPE("tuParm1") <> "O")
IF llCompareProperty AND TYPE("tuParm1") == "C"
	tuParm1 = UPPER(tuParm1)
ENDIF
llFound = .F.
lnFound = 0
*-- Determine direction to search
IF PCOUNT() < 3 OR TYPE("tnDirection") <> "N"
	tnDirection = 1
ENDIF
lnStart = IIF(tnDirection = 1, 1, ALEN(THIS.aChildren, 1))
lnEnd = IIF(tnDirection = 1, ALEN(THIS.aChildren, 1), 1)

*-- Scan this.aChildren[] looking for tuParm1.
FOR lnRow = lnStart TO lnEnd STEP tnDirection
	IF llCompareProperty
*-- Look for either class name, object name, or both
		IF EMPTY(tcProperty) OR tcProperty = "CLASS"
			llFound = UPPER(THIS.aChildren[lnRow, CHILD_CLASS]) ==
tuParm1
		ENDIF
		IF !llFound AND (EMPTY(tcProperty) OR tcProperty = "NAME")
			llFound = UPPER(THIS.aChildren[lnRow, CHILD_NAME]) == tuParm1
		ELSE
```

```
                    IF !llFound AND TYPE("tcProperty") == "C"
*-- Convert expression to UPPER if character
                           IF TYPE("tuParm1") == "C"
                                  lcCompareExpr = "UPPER(this.aChildren[lnRow, "
+;
                                                 ALLT(STR(CHILD_OBJECT)) + "]" + ;
                                                "." + tcProperty + ") == tuParm1 "
                           ELSE
                                  lcCompareExpr = "this.aChildren[lnRow, " +;
                                         ALLT(STR(CHILD_OBJECT)) + "]" + ;
                                         "." + tcProperty + " == tuParm1 "
                           ENDIF
                           llFound = EVAL(lcCompareExpr)
                    ENDIF
             ENDIF
       ELSE
             llFound = COMPOBJ(THIS.aChildren[lnRow, CHILD_OBJECT], tuParm1)
       ENDIF
       IF llFound
*-- Object was found. Increment the counter by 1
             lnFound = lnFound + 1
       ENDIF
ENDFOR

RETURN lnFound
```

To complete the process we still need to modify Codebook's CBaseForm a bit to take advantage of our new method. Modify CBaseForm found in the CForms class library. First, add a new property called nMaxInstances and set the value to 0. Next, go to the Init method and add the following bit of code to the very beginning:

```
*-- Adding checking for nMaxInstances. Limits
*-- the number of instances of a form.
IF this.WindowType <> WIN_MODAL AND this.nMaxInstances > 0
       IF TYPE("goApp.oForms") == "O"
             IF goapp.oForms.GetInstanceCount(this.class) >= this.nMaxInstances
                    = ERRORMSG('Maximum number of forms of this type reached!')
                    RETURN .F.
             ENDIF
       ENDIF
ENDIF
```

That's it! If nMaxInstances is set to anything other than 0 its value will be compared to the number of classes already instanced and return .F. if the maximum number has been exceeded. Not too bad for ten lines of code, huh?

Conclusion

Examining Codebook's collection classes allows us a better understanding of the framework. Collections are at the center of almost everything Codebook does. While the base collection classes are a good start there still is a lot of refining and extending that can be done.

Michael G. Emmons is president of Z-Buffer Data Technologies, a computer consulting firm, and a Microsoft Certified Professional in Visual Foxpro. He specializes in client/server solutions using Codebook and can be reached at mgemmons@netrunner.net.

SAVI Codebook Application Form

I would like to receive a ***FREE*** and ***UNLIMITED*** copy of the SAVI Codebook***. No more proof of purchase is required*.** I understand that the information displayed in **BOLDFACE** is mandatory and I will not receive my package if those portions of the application are left incomplete.

First Name _____

Middle Initial _____

Last Name _____

Company Name _____

Address _____

_____

City, State, Zip Code _____, _____ _____ - _____

e-mail Address _____

Work Phone Number _____

Work Fax Number _____

Home Phone Number _____

I would like the SAVI Codebook Framework for _____ VFP Version 5.0a
_____ VFP Version 3.0b

_____ I would like to receive the Rational Rose (V4.0.14) model for Codebook as well, VFP 5.0a version only.

Send completed application form to:

To: Software Assets of Virginia, Inc.
2109 Silbert Road
Kent Park
Norfolk, VA 23509-2126 USA
Attention: Free Codebook Offer

** - e-mail is the only method of delivery. If you do not provide a ***legible*** e-mail address, you will not receive the framework.