Clarion Magazine

This edition includes all articles, news items and blog posts from February 1 2011 to February 28 2011.

Clarion News

Read 16 Clarion news items.

Articles

Creating an SQL Query Class and Template, Part 3

February 1 2011

Phil Will wraps up the SQL Query class and template and explains the class methods in more detail.

Tip of the Week #9: Code Completion Window Width

February 2 2011

Sometimes you need just a little more real estate for the code completion window. It's easily done.

Webinar this Friday: If you can't test your code, how do you know it isn't crap?

February 8 2011

Is your code crap? How would you know? How can you easily test your code and have confidence that it's doing what you expect it to do? How can you automate your tests? This Friday Dave Harms will be the guest presenter at ClarionLive, where he'll answer these questions and others as applies unit testing to the ClarionLive Class Bash code.

Tip of the Week #10: Speed Up Source Navigation With Bookmarks

February 11 2011

If you ever find yourself jumping around between a few different places in source (or in the embeditor), you'll appreciate this week's tip.

MagGem: Validating Credit Card Numbers

February 16 2011

If you're handling credit card numbers, here's a way to check for bad data before you submit the card for processing.

MagGem: Backdoors and Other Tricks

February 17 2011

Let Keystate() give you administrator access to program features without requiring a special login.

Tip of the Week #11: The Task List

February 18 2011

The C7 Task List is yet another way to navigate around your source code. But there are a couple of things to watch out for.

Automagical Browse Refreshing

February 18 2011

Automatic browse refreshing may not be earthshattering news for many in the Clarion community, but it is one of John Morter's favorite techniques for simplifying complex browse interactions.

MagGem: Capitalize The Right Way

February 22 2011

Yes, Clarion has a CAP attribute. But it's not very smart. Here's the code you need to handle capitalization the right way.

MagGem: Everything You Ever Wanted To Know About Strings

February 24 2011

Does your app have more string data than you realize? Are there better ways to handle those strings? Almost certainly. If you deal with strings (and who doesn't) you really need to read this article.

Tip of the Week #12: Searching/Finding Files Using Redirection

February 27 2011

Both Clarion 6 and Clarion 7 make use of the redirection file to help you find your own source, but Clarion 6's approach still wins.

Highlighting Text With RTF The Easy Way, Part 1

February 28 2011

Dave Harms builds on a MagGem by Stephen Bottomley, and creates a class that makes it easy to display color highlighted text using the RTF control. Part 1 of 2.

Clarion News

Clarion QuickBooks Connect 1.25

Clarion QuickBooks Connect 1.25 is available for download.

Posted February 4 2011 (permanent link)

Clarion QuickBooks Connect Video Training

Video training is available for Clarion QuickBooks Connect. The first two videos show how to import file definitions from the qbc_Helper program.

Posted February 4 2011 (permanent link)

JS Mailmerge for Clarion7.3.7900

Jeff Slarve has updated the JS MailMerge templates for C7.3.7900. Please private email Jeff if you are a licensed user and need the password.

Posted February 4 2011 (permanent link)

ClarionEditorContextHelp AddIn

Brahn Patridge's has released an early version of an addin that adds context help to the IDE. Download the addin, install it with the Addin Manager, and start using Shift-F1 for context help.

Posted February 5 2011 (permanent link)

Clarion 7.3.7949 Released

SoftVelocity has released Clarion 7 build 7949. This release includes a number fixes for IDE crashes, TXA exporting, and the Report Writer.

Posted February 5 2011 (permanent link)

DMC Flash Special - Up To 50% Off

Because of an unexpected family emergency JP needs to raise cash very quickly and is holding a DMC Flash Sale. From Saturday Feb 12 until Sunday Feb 20 2011 DMC is 50% off the regular price. From Monday Feb 21 until Sunday, Feb 26 2011 DMC is 25% off, and from Monday Feb 27 till Tuesday Feb 28 2011 DMC is 10% off. Discounts end at midnight GMT on the date specified and apply to updates or renewals. Use this coupon code on the SWREG payment portal: DMC-45K2D46627. The prices onsite do not reflect the discounts

- they are applied only when you purchase with this coupon code.

Posted February 12 2011 (permanent link)

Clarion 7.3.7989 Released

A new build of Clarion 7.3 is now available. Changes include fixes to RUN and #RUN, TXA exporting, and the visual appearance of some controls.

Posted February 15 2011 (permanent link)

iQ-XML, iQ-Sync Updated

New versions of iQ-XML and iQ-Sync are now available. iQ-Xml is compiled in C7.3.7949 and has a fix to handling XML comments inside XML Comments, within valid XML sections. iQ-Sync is compiled in C7.3.7949 and has corrections to handle backing up more than 32,768 files (SHORT) as well as enhancements to the ZIP options.

Posted February 18 2011 (permanent link)

IQ-SQL Beta Released

A beta of IQ-SQL is now available. This release works only with Sybase and PostgreSQL. This is a developer tool, but designed to plug into a Clarion application. This program has been in production for more than 700 users for the last three years on Sybase SQL Anywhere. It hasn't been used at all for PostGreSQL, so that is where the testing is needed. For PostGres, the the ANSI or UNICODE driver must be installed. This is a first release of this product outside of InnQuest. To get started, look at the online help, SQL Syntax to learn how to use the Extensions or use the Wizard to write them for you. Automatic JOINS is not documented yet. Robert Paresi isn't looking for enhancement requests, just testing. Do not install this at an end-user site, nor do you have a license to do so. This is for testing purposes only.

Posted February 18 2011 (permanent link)

CHT Videos

Gus has posted new videos to illustrate and teach about some of the new work going on at CHT. Topics include: CHT Web Scripter Example Exercises; CHT Web Servers and Blackberry Smartphones, Blackberry PlayPad; CHT Compile Manager For Clarion 7.

Posted February 18 2011 (permanent link)

SetupBuilder 7.3 Build 3228

LinderSoft has released SetupBuilder 7.3 Build 3228. This release is available, free of charge, to all SetupBuilder customers who have an active SetupBuilder maintenance and support plan subscription.

Posted February 18 2011 (permanent link)

Clarion QB Connect Debugging Video

Bob Roos has added a new video to the collection for Clarion QB Connect. It shows the steps to add the debugging option and what output that produces.

Posted February 18 2011 (permanent link)

Charles Edmonds Takes Over ClarionDesktop

StrategyOnline has announced it is handing off the ClarionDesktop product to Charles Edmonds of LANSRAD.

Posted February 18 2011 (permanent link)

J-Spell 2.21

J-Spell 2.21 is now available. this release includes a regression fix related to address handling.

Posted February 18 2011 (permanent link)

Clarion 7.3.7995 Released

SoftVelocity has released Clarion 7.3.7995, with a fix for a regression introduced in the previous release that caused an IDE hang related to the embeditor.

Posted February 18 2011 (permanent link)

RPM/AFE Subscription Intro Pricing Ends March 12 2011

On 01-Jan-2009 Lodestar Software adopted an annual subscription plan for RPM and AFE. Introduction pricing for current users, who are not already enrolled, will end March 12.

Posted February 25 2011 (permanent link)

The ClarionMag Blog

Creating an SQL Query Class and Template, Part 3

By Phil Will

Posted February 1 2011

In Part I of this series showed a process for developing a class shell and Control Template that are ABC-compliant, and I developed a general specification based on an example of a procedure-based set of SQL Query controls. These included code from a procedure with a browse box with a set of query controls originally developed by Bob Huff.

In Part II I showed an approach to handling the many controls and events in the context of a class library. In this part I will look at modifying the procedure code into a generic class library.

So far I have created only one property for the class – a queue of the control field field equates that includes a field number identifier and a string field. This queue is used to add specific function calls to the query filter ('BETWEEN', 'CONTAINS', etc.). I also created some methods for handling events – TakeEvent, TakeAccepted, TakeDropEvent. I've also crreated a wrapper template that will instantiate an instance of the class and populate the controls on a window. And finally, there is a test application in place with the controls populated and class instantiated.

Adding property references

To get started with the next phase, it is apparent from reviewing the procedure code that certain properties would be useful for the class. A reference to the BrowseClass used by the related browse control will clearly have some helpful items – the view and methods for filtering and refreshing the browse among them. References to the Browse Queue and QueryBuilder field will be needed. Since part of the spec calls for loading and saving queries, a reference to the query file manager will be helpful along with references to its keys and fields. A queue of fields that populate the drop down list of file fields that can be used in the query is needed – I prefer this to the string FROM property built in the procedure example. The spec also calls for lists of files and fields that should be excluded from the queue of filter fields.

The queues all have TYPE attribute and will be created in the Construct method and Disposed in the Destruct method. Hopefully the names are self explanatory.

```
QueryFieldQT        QUEUE,TYPE
QueryField            CSTRING(250)
                    END
ExcludeFilesQT      QUEUE,TYPE
ExcludeFile            STRING(250)
                    END
ExcludeFieldsQT     QUEUE,TYPE
ExcludeField          STRING(80)
                    END
QueryQT             QUEUE,TYPE
QueryName             STRING(250)
QueryID               ANY
                    END
```

The properties also include a group that can be passed to the class in the init method that identifies the FileManager for the saved queries and the field used for categorizing the queries.

```
SavedQueryGT        GROUP,TYPE
FM                    &FileManager
Category              ANY
                    END
```

Class properties

References for these are then added to the class properties list.

```
!-- Query controls
FieldQ          &FieldQT
!-- Browse Class.
BC              &BrowseClass
ListQueue       &QUEUE
!-- Query
Query           &STRING
QueryFieldQ     &QueryFieldQT
ExcludeFilesQ   &ExcludeFilesQT
ExcludeFieldsQ  &ExcludeFieldsQT
!-- Saved Query File References
QueryG          GROUP(SavedQueryGT)
                END
QueryfileIDKey  &KEY
QueryFileID     ANY
QueryFileCategoryKey    &KEY
QueryfileCategory   ANY
QueryFileName   ANY
QueryFileQuery  ANY
```

```
!—For drag and drop from the query field list.
QueryFieldRegion LONG
```

Some of the references can easily be passed as parameters in the `Init` method. Others can be handled using view and file properties. The next step is to modify the `Init` method to take the Browse Class, the Browse Queue, and the `SavedQuery` Group as parameters.

The template code to generate and assign references to the `SavedQuery` Group and identify the browse class object and queue requires some prompts and some additions to the % AfterOpeningWindow embed. Note that this is a Legacy and ABC embed which would be important if I were developing a template that applied to both ABC and Legacy applications.

These are the instance prompts:

```
#TAB('General')
  #BUTTON('Browse Info')
    #PROMPT('Browse Class',@s20),%SqlBrowse,REQ
    #PROMPT('Query FieldD',FIELD),%SqlQueryFld,REQ
    #PROMPT('Browse Queue',@S80),%BrowseListQueue,REQ
  #ENDBUTTON
  #BUTTON('Query Info')
    #PROMPT('Saved Query File',FILE),%SqlSavedQueryFile,REQ
    #PROMPT('Saved Query Category',@s20),%SqlSavedQueryCategory,REQ,DEFAULT('%Procedure')
    #BUTTON('Exclude Files'),MULTI(%ExcludeFiles,%ExcludeFile)
      #PROMPT('File to Exclude:',FILE),%ExcludeFile,REQ
    #ENDBUTTON
    #BUTTON('Exclude Fields'),MULTI(%ExcludeFields,%ExcludeField)
      #PROMPT('Field to Exclude:',FIELD),%ExcludeField,REQ
    #ENDBUTTON
  #ENDBUTTON
#ENDTAB
```

And this is the generated code:

```
#AT(%AfterOpeningWindow)
#!-----------------------------------------------------------------------------
_QueryG.FM &= %QueryFM
_QueryG.Category=%SqlSavedQueryCategory
%ThisObjectName.Init(%SqlBrowse,%SqlQueryFld,%BrowseListQueue,_QueryG)
SELF.AddItem(%ThisObjectName.WindowComponent)
#ENDAT
#!-----------------------------------------------------------------------------
#AT(%PDSQLFILTERMethodCodeSection,%ActiveTemplateInstance),PRIORITY(2500),WHERE(%pClassMethod='Init')
#!-----------------------------------------------------------------------------
  #FOR(%ExcludeFiles)
    #FIND(%File,%ExcludeFile)
    #IF(%FileName)
SELF.AddExcludeFile(%FileName)
    #ELSE
SELF.AddExcludeFile('%ExcludeFile')
    #ENDIF
  #ENDFOR
  #FOR(%ExcludeFields)
SELF.AddExcludeField('%ExcludeField')
  #ENDFOR
#ENDAT
```

Using the test application, I can make some template entries, look at the generated code, and compile. This still won't have any functionality, but I'm making progress...

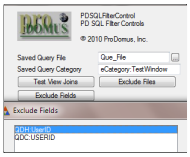

Figure 1. Sample template entries.

Checking the generated code, the following generated code can be found in the `WindowManager.Init` embeds. Note that `eCategory` has itemized equates in the translation file as a convenient place for the developer to add and edit them.

```
_QueryG.FM &= Access:Que_File
  _QueryG.Category=eCategory:TestWindow
  PDSQLFilterC4.Init(BRW2,QueryBuilder,Queue:Browse,_QueryG)
  SELF.AddItem(PDSQLFilterC4.WindowComponent)
```

In the query class instance `Init` method embed has the following generated:

```
PDSQLFilterC4.Init PROCEDURE(BrowseClass pBC,*STRING pQuery,*QUEUE pListQueue,SavedQueryGT pQueryG)
  CODE
  SELF.AddExcludeField('QDH:UserID')
  SELF.AddExcludeField('QDC:USERID')
  PARENT.Init(pBC,pQuery,pListQueue,pQueryG)
```

SetAlerts

Earlier I created a SetAlerts method and called this from the WindowComponent SetAlerts. This is a good place to set any runtime properties. At this point at runtime, the class has been initialized and the window is open.

While implementing Drag and Drop from the Query Field Drop List, I discovered that I could not drop from the drop list to the QueryBuilder field; instead I had to create a region with a DragId. Rather than making the region a control that is part of the control template, I added a property for the region's field equate (QueryFieldRegion) and create the control in the SetAlerts method. I also use the FieldQ.FieldFEQ to set drag and drop IDs for the list, the text field, and the QueryBuilder fields.

```
!-----------------------------------------------------------------------------
PDSQLFilterCT.SetAlerts PROCEDURE()
!-----------------------------------------------------------------------------
  CODE
    !-- Create the Drag region above the QueryField Drop List.
    IF ~SELF.GetFieldFEQ(eFld:QueryField)
      SELF.QueryFieldRegion=CREATE(0,CREATE:region,SELF.FieldQ.FieldFEQ{PROP:Parent})
      SELF.QueryFieldRegion{PROP:Width}=SELF.FieldQ.FieldFEQ{PROP:Width}
      SELF.QueryFieldRegion{PROP:XPos}=SELF.FieldQ.FieldFEQ{PROP:Xpos}
      SELF.QueryFieldRegion{PROP:YPos}=SELF.FieldQ.FieldFEQ{PROP:Ypos}-12
      SELF.QueryFieldRegion{PROP:Height}=12
      SELF.QueryFieldRegion{PROP:TRN}=TRUE
      SELF.QueryFieldRegion{PROP:DragID}='QUERYFIELD'
      UNHIDE(SELF.QueryFieldRegion)
    END
    !-- Set Other Drag and Drop ID's
    SELF.BC.ILC.GetControl(){PROP:DragId}='PDFILTERLIST'
    IF ~SELF.GetFieldFEQ(eFld:QueryField)
      SELf.FieldQ.FieldFEQ{PROP:From} = SELF.QueryFieldQ
    END
    IF ~SELF.GetFieldFEQ(eFld:QueryAction)
      SELF.GetQueryActions
    END
    IF ~SELF.GetFieldFEQ(eFld:Alpha)
      SELf.FieldQ.FieldFEQ{PROP:DropID}='PDFILTERLIST'
    END
    IF ~SELF.GetFieldFEQ(eFld:QueryBuilder)
      SELf.FieldQ.FieldFEQ{PROP:DropID}='QUERYFIELD'
    END
```

Note that the BC property is the BrowseClass for the parent browse and its "ILC" interface GetControl method returns the field equate for the browse list. The QueryField Queue also gets assign to the FROM property of the Query Field Drop List.

This also introduces a new method for getting retrieving the FieldQ record that contains a specific Field Number. Like IF ERRORCODE(), this method returns a positive value if there is an error getting the FieldQ record or if the FieldQ record has no FieldFEQ value.

```
!-----------------------------------------------------------------------------
PDSQLFilterCT.GetFieldFEQ PROCEDURE(SIGNED pFld)
!-----------------------------------------------------------------------------
RV                          BYTE
  CODE
    SELF.FieldQ.FieldNo=pFld
    GET(SELF.FieldQ,SELF.FieldQ.FieldNo)
   !-- return true if a valid field control value is found
    RETURN CHOOSE(SELF.FieldQ.FieldFEQ=0 OR ERRORCODE(),TRUE,FALSE)
```

One advantage of the methodology being used here is that if controls were removed from the window, the class library would not make any assignments for that control. The control template uses a major piece of window real estate. It is possible to remove everything but the QueryBuilder, Execute, and Reset controls and still retain query functionality. You might also derive a class that puts most of the controls on a separate window.

Extracting view, file and field information

When Bob Huff developed the original procedures and routines he knew what files were in the view, external file names, and file prefixes. An example of this is sown in the GetQueryFields routine.

```
GetQueryFields Routine
GetQueryFields      ROUTINE
  DATA
FieldCount  LONG
Counter LONG
FromString  STRING(5000)
TempString  STRING(250)
PrefixLen   LONG
  CODE

    FieldCount = QD_Hold{PROP:Fields}
    PrefixLen = LEN('QDH:')
    LOOP FieldCount TIMES
      Counter += 1
      TempString = WHO(QDH:Record,Counter)
      FromString = CLIP(FromString) & 'QD_Hold.[' & TempString[PrefixLen + 1 : LEN(CLIP(TempString))] & ']|'
    END

    CLEAR(Counter)
    FieldCount = ID_Lot{PROP:Fields}
```

```
    PrefixLen = LEN('ID_L:')
    LOOP FieldCount TIMES
      Counter += 1
      TempString = WHO(ID_L:Record,Counter)
      FromString = CLIP(FromString) & 'ID_Lot.[' & TempString[PrefixLen + 1 : LEN(CLIP(TempString))] & ']|'
    END

    IF CLIP(FromString) THEN FromString = FromString[1 : LEN(CLIP(FromString)) - 1].
    ?LOC:QueryField{PROP:From} = UPPER(CLIP(FromString))
```

AddQueryFields

In the class, all that is known is the name of the view. The information will have to be extracted using properties. I also specified that I wanted to put the values in a queue rather than creating a long `FROM` string and that I wanted the developer to be able to exclude specific files and fields. The routine is helpful in showing a workable use of `WHO` and `PROP:Fields`, something that can be confusing. It also shows how the fields should be formatted for use in SQL.

I changed the method name to `AddQueryFields` (my preference), and started with three lines of code to get the number of files in the view and then add fields from each file.

```
!-----------------------------------------------------------------------------
PDSQLFilterCT.AddQueryFields PROCEDURE()
!-----------------------------------------------------------------------------
FileCount                       LONG
FieldCount                      LONG
FieldLabel                      STRING(255)
Counter                         LONG
FieldCounter                    LONG
TempString                      STRING(250)
ThisLabel                       STRING(250)
FileLabel                       STRING(250)
ThisPrefixLen                   LONG
ThisFile                        &FILE
PrefixFound                     BYTE
  CODE
    FileCount = SELF.BC.View{PROP:Files,0}
    LOOP Counter=1 TO FileCount
      DO AddFiles
    END
```

To break this method down into workable blocks, the first part of the method simply gets the file count from the view and the calls a routine to take it from there.

```
AddFiles            ROUTINE
  ThisFile &= SELF.BC.View{PROP:File,Counter}
  SELF.ExcludeFilesQ.ExcludeFile = UPPER(ThisFile{PROP:Name})
  !-- Check exclude file list.
  GET(SELF.ExcludeFilesQ,SELF.ExcludeFilesQ.ExcludeFile)
  IF NOT ERRORCODE()
    EXIT
  END
  DO AddFile
```

This routine gets the file and its name and then checks to see if it is an exclude file. If it is an exclude file, processing returns to the loop.

```
AddFile ROUTINE
  FieldCount = ThisFile{PROP:Fields}
  ThisLabel = ThisFile{PROP:Name}
  IF ThisLabel=''
    ThisLabel = ThisFile{PROP:label,0}
  END
  SELF.ParseLabel(ThisLabel)
  IF Counter=1
    SELF.AddJoin(ThisFile,'')
  END
  ThisFile{PROP:Alias}=Thislabel
  DO AddFileFields
```

The `AddFile` routine gets the label for the file, calling another method for parsing the name string. This basically removes `dbo.` if it's there.

The `AddJoin` call was developed later when coding the `CheckQuery` method. It adds files and their relation statements to the `JOIN` statement in the view used to check queries. The first file has no join statement. The `PROP:Alias` is needed by SQL to recognize file names rather the default aliases of "a,b,c,etc." I learned that the hard way as it was not in any of the sample code.

```
AddFileFields           ROUTINE
  PrefixFound=FALSE
  LOOP FieldCounter=1 TO FieldCount
    FieldLabel=ThisFile{PROP:Label,FieldCounter}
    !-- Check for excluded field
    SELF.ExcludeFieldsQ.ExcludeField = UPPER(FieldLabel)
    GET(SELF.ExcludeFieldsQ,SELF.ExcludeFieldsQ.ExcludeField)
    IF NOT ERRORCODE()
      EXIT
    END
    !-- Add to list of query fields and queue of prefix cross references.
    CLEAR(TempString)
```

```
    TempString=ThisFile{PROP:Name,FieldCounter}
    IF TempString=''
      TempString = ThisFile{PROP:Label,FieldCounter}
    END
    IF TempString<>''
      IF ~PrefixFound
        PrefixFound=TRUE
        ThisPrefixLen=INSTRING(':',FieldLabel,1,1)
        SELF.DebugOut('PrefixLen: '&ThisPrefixLen,'Filter.AddQueryFields')
        IF ThisPrefixLen
          SELF.LabelQ.Prefix=FieldLabel[1: ThisPrefixLen]
          SELF.LabelQ.Label=ThisLabel
          SELF.DebugOut('Add Prefix: '&SELF.LabelQ.Prefix,'Filter.AddQueryFields')
          ADD(SELF.LabelQ,SELF.LabelQ.Prefix)
        END
      END
      SELF.ParseField(TempString,ThisLabel)
      SELF.AddQueryField(CLIP(TempString))
    END
  END
```

The `AddFileFields` routine, which does several tasks, is a good candidate for further factoring, but it is what it is. It first check s whether the field is in the list of exclude fields and exits if it finds it there. It then gets the field's name. In later development of dragging fields from the browse, I found the need for a new queue (`labelQ`) that related the file prefix with the file name. This queue was created the same way as other, coding a queue with the `TYPE` attribute, adding a property for the queue, and then creating and disposing it in the `Construct` and `Destruct` methods. The routine then calls a method (`ParseField`) to format the field and combine it with the label. Finally, it calls a method to add it the Query Field queue. The developer could use this method to add other fields or to abort the add if there were conditions where the field should not be added. An example would be a field that is available with only certain security rights.

The DropData routine

As in the `GetQueryFields` routine, the `DropData` routine (which is called when dragging from the browse to the text field (`Alpha`)) assumes knowledge of the specific browse and the prefixes. It gets the column and row choices and changes to field choice to SQL syntax. It also displays dates and times in a readable format and assigns the results to the `QueryField` and `Alpha` fields and sets the `QueryAction` field to "=". It uses a local queue (`QCopy`) to get the field labels.

```
DropData            ROUTINE
  DATA
RowChoice   LONG
ColumnChoice    LONG
FieldChoice STRING(250)
PrefixLen   LONG
QCopy   QUEUE,BINDABLE
HSITE     LONG
HPalletNum    LONG
HHoldTicket   LONG
HPONumber LONG
HDate     LONG
HTime     LONG
HStatus   LONG
LType     LONG
LVendorID LONG  !Bob
HOraclePallet LONG
LStatus   LONG
LSite     LONG
LPalletNum    LONG
        END
ValPic  STRING(20)
  CODE
    RowChoice = ?HoldList{PROPLIST:MouseDownRow}
    ColumnChoice = ?HoldList{PROPLIST:MouseDownField}
    GET(Queue:Browse,RowChoice) ; IF ERRORCODE() THEN EXIT.
    LOC:Alpha = WHAT(Queue:Browse,ColumnChoice)
    ValPic = ?HoldList{PROPLIST:Picture,ColumnChoice}
    CASE UPPER(ValPic[2])
    OF 'S'
      LOC:Alpha = '''' & CLIP(LOC:Alpha) & ''''
    OF 'D'
      LOC:Alpha = 'DATE<' & FORMAT(LOC:Alpha,@D01) & '>'
    OF 'T'
      LOC:Alpha = 'TIME<' & FORMAT(LOC:Alpha,@T04) & '>'
    END
    FieldChoice = WHO(QCopy,ColumnChoice)
! stop(FieldChoice)
    PrefixLen = LEN('f')
    LOC:QueryField = UPPER(FieldChoice[PrefixLen + 1 : LEN(CLIP(FieldChoice))]) & ']'

    IF UPPER(FieldChoice[1]) = 'H'
      LOC:QueryField = 'QD_HOLD.[' & CLIP(LOC:QueryField)
    ELSE
      LOC:QueryField = 'ID_LOT.[' & CLIP(LOC:QueryField)
    END
    LOC:QueryAction = '='
```

```
    DISPLAY(?LOC:QueryField)
    DISPLAY(?LOC:Alpha)
```

TakeDropEvent

The `TakeDropEvent` method renames the original `DropData` routine and handles both the drop to the `Alpha` field from the browse and the drop to the `QueryBuilder` field from the `QueryField` drop list. The handling of these two drops is put into separate routines to make it more readable. The `CopyQ` has been eliminated as file labels and field names can be extracted from the browse queue properties.

```
!-----------------------------------------------------------------------------
PDSQLFilterCT.TakeDropEvent PROCEDURE()
!-----------------------------------------------------------------------------
RowChoice                    LONG
ColumnChoice                 LONG
FieldChoice                  STRING(250)
PrefixLen                    LONG
ValPic                       STRING(20)
TempString                   STRING(250)
lAlpha                       ANY
  CODE
    SELF.DebugOut('DragID: '&DRAGID(),'Filter.TakeDropEvent')
    UPDATE()
    CASE DRAGID()
    OF eBrowseDragID
      DO GetBrowseItem
    OF eQueryFieldDragID
      DO GetQueryFieldItem
    END
```

The GetBrowseItem routine

The new routine echoes the coding of the original `DropData` routine, using properties rather than routine specific information. Needed values are retrieved from the browse class (`BC`) `ListQueue`. Because the file name is not available from the `ListQueue`, I use the previously populated `LabelQ` to retrieve the file name by using the prefix as a lookup.

Some of the debug code has been left in here to illustrate my testing approach.

```
GetBrowseItem       ROUTINE
    RowChoice = SELF.BC.ILC.GetControl(){PROPLIST:MouseDownRow}
    ColumnChoice = SELF.BC.ILC.GetControl(){PROPLIST:MouseDownField}
    GET(SELF.ListQueue,RowChoice)
    IF ERRORCODE()
      RETURN
    END
    lAlpha = WHAT(SELF.ListQueue,ColumnChoice)
    ValPic = SELF.BC.ILC.GetControl(){PROPLIST:Picture,ColumnChoice}
    CASE UPPER(ValPic[2])
    OF 'S'
      lAlpha = '''' & CLIP(LAlpha) & ''''
    OF 'D'
      lAlpha = 'DATE<<' & FORMAT(lAlpha,@D01) & '>'
    OF 'T'
      lAlpha = 'TIME<<' & FORMAT(LAlpha,@T04) & '>'
    END
    IF ~SELF.GetFieldFeq(eFld:Alpha)
      CHANGE(SELF.FieldQ.FieldFEQ,lAlpha)
    END
    FieldChoice = WHO(SELF.ListQueue,ColumnChoice)
    SELF.DebugOut('FieldChoice: '&CLIP(FieldChoice),'Filter.TakeDropEvent')
    PrefixLen = INSTRING(':',FieldChoice,1,1) !LEN('f')
    IF PreFixLen
      SELF.LabelQ.Prefix=FieldChoice[1 : PrefixLen]
      GET(SELF.LabelQ,SELF.LabelQ.Prefix)
      IF NOT ERRORCODE()
        SELF.ParseField(FieldChoice,SELF.LabelQ.Label)
      ELSE
        SELF.DebugOut('Error getting label: '&ERROR(),'Filter.TakeDropEvent')
        EXIT
      END
    END
    SELF.DebugOut('Field: '&CLIP(FieldChoice),'Filter.TakeDropEvent')
    IF ~SELF.GetFieldFeq(eFld:QueryField)
      CHANGE(SELF.FieldQ.FieldFEQ,FieldChoice) | !UPPER(FieldChoice[PrefixLen + 1 : LEN(CLIP(FieldChoice))]) & ']'
    ELSE
      SELF.DebugOut('Error getting FieldFEQ','Filter.TakeDropEvent')
    END
    IF ~SELF.GetFieldFEQ(eFld:QueryAction)
      CHANGE(SELF.FieldQ.FieldFEQ, '=')
    END
    DISPLAY
```

The GetQueryFieldItem routine

This routine appends the contents of the `QueryField` to the `QueryBuilder` field by first getting the field equate for the `QueryField` and then calling the `BuildAction` method which handles the concatenation.

```
GetQueryFieldItem   ROUTINE
   IF ~SELF.GetFieldFEQ(eFld:QueryField)
    SELF.BuildAction(CONTENTS(SELF.FieldQ.FieldFEQ))
    DISPLAY(SELF.FieldQ.FieldFEQ)
   END
```

Building and checking the query

The original `BuildAction` procedure has the following code:

```
BuildAction         PROCEDURE(pAct)
  CODE
    IF CLIP(LOC:QueryBuilder)
      LOC:QueryBuilder = CLIP(LOC:QueryBuilder) & ' ' & pAct
    ELSE
      LOC:QueryBuilder = pAct
    END

    DISPLAY(?LOC:QueryBuilder)

    IF CheckQuery(LOC:QueryBuilder) THEN DO GoGreen ELSE DO GoRed.
    RETURN
```

The `BuildAction` method is essentially the same as the local procedure in the example. The calls to `GoGreen` and `GoRed` have been moved to the `CheckQuery` method to be discussed below as they need to be implemented whenever the `CheckQuery` method is called. The `CheckQuery` method also uses the `SELF.Query` property of the class, so the parameter is no longer needed. The assignment to `SELF.Query` has been reduced to one line of code.

```
!-----------------------------------------------------------------------------
PDSQLFilterCT.BuildAction PROCEDURE(STRING pAction)
!-----------------------------------------------------------------------------
  CODE
    SELF.Query=LEFT(CLIP(SELF.Query) & ' ' & pAction)
    DISPLAY
    SELF.CheckQuery()
```

Each time the `QueryBuilder` field (`SELF.Query` property) is changed, it is checked by executing an SQL statement. The sample procedure uses a locally declared view and a hand coded `SELECT` query, returning true or false depending on whether there is an error.

Here is a generic view:

```
QCheckView          VIEW(A_GenericValues)
                      PROJECT(GV:String1)
                    END
```

Because the query is not actually executing (`SET NOEXEC ON`), the new `CheckQuery` method uses the existing Browse view rather than creating a separate test view. The `GoGreen` and `GoRed` procedures are called depending on whether the query succeeds or fails. A new class property is introduced to display the Query Error when an attempt is made to actually execute the query and the test fails.

```
CheckQuery          PROCEDURE(pQuery)
LocalReturnValue      BYTE
  CODE
    OPEN(QCheckView)
    QCheckView{PROP:SQL} = 'SET NOEXEC ON SELECT * FROM ' & NAME(QD_Hold) & |
      ' JOIN ' & NAME(ID_Lot) & ' ON QD_Hold.[PalletNum] = ID_Lot.[PalletNum] ' & |
      ' WHERE ' & CLIP(DecodeQuery(pQuery)) & ' SET NOEXEC OFF'
    IF ERRORCODE() = 0 THEN LocalReturnValue = TRUE.
    CLOSE(QCheckView)
    DO EndLocalProcedure
EndLocalProcedure   ROUTINE
  RETURN(LocalReturnValue)
```

The `CheckQuery` procedure calls a `DecodeQuery` procedure that converts the Date and Time entries in the query to SQL formats.

```
CheckQuery          PROCEDURE(pQuery)
LocalReturnValue      BYTE
  CODE
    OPEN(QCheckView)
    QCheckView{PROP:SQL} = 'SET NOEXEC ON SELECT * FROM ' & NAME(QD_Hold) & |
      ' JOIN ' & NAME(ID_Lot) & ' ON QD_Hold.[PalletNum] = ID_Lot.[PalletNum] ' & |
      ' WHERE ' & CLIP(DecodeQuery(pQuery)) & ' SET NOEXEC OFF'
    IF ERRORCODE() = 0 THEN LocalReturnValue = TRUE.
    CLOSE(QCheckView)
    DO EndLocalProcedure
EndLocalProcedure   ROUTINE
  RETURN(LocalReturnValue)
```

The `DecodeQuery` method and local routine are exactly the same except for doubling the left angle brackets.

```
DecodeQuery         PROCEDURE(pCodedQuery) ! From local procedure.
LocalReturnValue      STRING(10000)
StringPos             LONG
EndBracket            LONG
CodeLen               LONG

  CODE
```

```
    CodeLen = LEN('DATE<')
  ! Fix Dates
    LOOP
      StringPos = INSTRING('DATE<',pCodedQuery,1,1)
      EndBracket = INSTRING('>',pCodedQuery,1,StringPos)
      IF StringPos AND EndBracket
        pCodedQuery = pCodedQuery[1 : StringPos - 1] & |
          DEFORMAT(pCodedQuery[StringPos + CodeLen : EndBracket - 1],@D01) & |
          pCodedQuery[EndBracket + 1 : LEN(CLIP(pCodedQuery))]
      ELSE
        BREAK
      END
    END

  ! Fix Times
    CodeLen = LEN('TIME<')
    LOOP
      StringPos = INSTRING('TIME<',pCodedQuery,1,1)
      EndBracket = INSTRING('>',pCodedQuery,1,StringPos)
      IF StringPos AND EndBracket
        pCodedQuery = pCodedQuery[1 : StringPos - 1] & |
          DEFORMAT(pCodedQuery[StringPos + CodeLen : EndBracket - 1],@D04) & |
          pCodedQuery[EndBracket + 1 : LEN(CLIP(pCodedQuery))]
      ELSE
        BREAK
      END
    END

    LocalReturnValue = pCodedQuery
    RETURN(pCodedQuery)
```

The `GetJoin` method returns a string with all the files required by the query and their join statements. It requires a new queue of file names and their join statements, a method to add files based on template entries, and the `GetJoin` method to build the string from the queue. The queue property is created and disposed in the `Construct` and `Destruct` methods. Shown below are the queue definition and the two methods.

```
ViewQT              QUEUE,TYPE
ViewFile              &FILE
ViewJoin              CSTRING(255)
                    END
!-----------------------------------------------------------------------------
PDSQLFilterCT.AddJoin PROCEDURE(FILE pFile,STRING pJoin)
!-----------------------------------------------------------------------------
  CODE
    SELF.ViewQ.ViewFile &= pFile
    SELF.ViewQ.ViewJoin=pJoin
    ADD(SELF.ViewQ)
!-----------------------------------------------------------------------------
PDSQLFilterCT.GetJoin PROCEDURE()
!-----------------------------------------------------------------------------
RV                      STRING(5000)
  CODE
    LOOP I#=1 TO RECORDS(SELF.ViewQ)
      GET(SELF.ViewQ,I#)
      RV = LEFT(CLIP(RV)&CHOOSE(SELF.ViewQ.ViewJoin<>'',' JOIN','')&' '&NAME(SELF.ViewQ.ViewFile)&' '&CLIP(SELF.Vi
    END
    SELF.DebugOut('Join: '&CLIP(RV),'Filter.GetJoin')
    RETURN CLIP(RV)
```

The template prompts use a button with the `MULTI` attribute. Note that the class code adds the primary file to the queue in the `AddQueryFields` method.

```
     #BUTTON('Test View Joins'),MULTI(%Joins,%joinFile)
       #PROMPT('Join File',FILE),%JoinFile,REQ
       #PROMPT('Join Expression',TEXT),%JoinExpression
     #ENDBUTTON
```

The template then adds the joins at the end of the init method.

```
#!-----------------------------------------------------------------------------
#AT(%PDSQLFILTERMethodCodeSection,%ActiveTemplateInstance),PRIORITY(7500),WHERE(%pClassMethod='Init')
#!-----------------------------------------------------------------------------
  #FOR(%FilterField)
SELF.AddField(%FilterFieldNo,%FilterField)
  #ENDFOR
  #FOR(%Joins)
SELF.AddJoin(%JoinFile,'%JoinExpression')
  #ENDFOR
#ENDAT
```

Here is a sample entry.

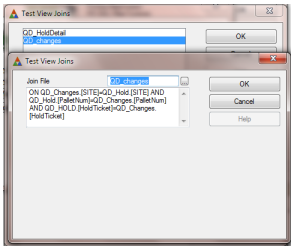

Figure 2. Template entries for joins.

Executing queries

The `ExecQuery` routine in procedure code checks the query, then sets the `PROP:SqlFilter`, and calls two browse class method to redisplay the query.

```
ExecQuery Routine
ExecQuery           ROUTINE
  IF ~CheckQuery(LOC:QueryBuilder)
    MESSAGE('Your query statement contains errors.  Please correct these errors first.','Query Not Executed..')
    EXIT
  END

  BRW4::View:Browse{PROP:SQLFilter} = CLIP(DecodeQuery(LOC:QueryBuilder))
  BRW4.ResetFromBuffer
  BRW4.PostNewSelection
```

The `ExecQuery` method makes a couple of small changes. The first is to display the `QueryError` if one exists after a failed call to `CheckQuery`.

```
!-----------------------------------------------------------------------------
PDSQLFilterCT.ExecQuery PROCEDURE()
!-----------------------------------------------------------------------------
  CODE
    IF ~SELF.CheckQuery()
      MESSAGE(SELF.TranslateString(eExecQueryMsga)&':<13,10,13,10>'&|
        CLIP(SELF.QueryError)&'<13,10><13,10>  '&SELF.TranslateString(eExecQueryMsgB),|
        SELF.TranslateString(eExecQueryMsgTitle),|
        ICON:Exclamation)
      CLEAR(SELF.QueryError)
    ELSE
      SELF.SetSqlFilter()
      SELF.BC.ResetFromBuffer
      SELF.BC.PostNewSelection
    END
```

The second is to translate the strings in the `MESSAGE`. Note that PD Translator Plus will automatically translate messages using `PROP:MessageHook`, but would not translate a variable message like this one. Because it is a variable message, the message text is put into an equate which in turn is put in the Translation (TRN) file. These strings are then put into a `TranslateString` method call.

```
eExecQueryMsgA      EQUATE('Your query statement contains errors')
eExecQueryMsgB      EQUATE('Please correct these errors first.')
eExecQueryMsgTitle  EQUATE('Query Not Executed..')
```

A final change is to call a new `SetSqlFilter` method. Rather than assigning the filter string the view `SQLFilter` property, wiping out any filters that may exist within the browse class, it uses the browse classes `SetFilter` method with a second parameter to identify the filter name. The `ResetFromBuffer` browse class method builds the filter internally.

```
!-----------------------------------------------------------------------------
PDSQLFilterCT.SetSQLFilter PROCEDURE()
!-----------------------------------------------------------------------------
  CODE
    IF SELF.Query<>''
      SELF.BC.SetFilter('(SQL(' & SELF.DecodeQuery(SELF.Query) & '))','PDSQLFILTER')
    ELSE
      SELF.BC.SetFilter('','PDSQLFILTER')
    END
```

Saving and loading queries

In the `Init` method I have covered passing the `FileManager` for saved queries and the field used for query categories. Some additional minimum properties are necessary for loading and saving queries. The class library assumes a simple structure for this file and then extracts needed information from file, key, and file record. It also adds virtual methods that that can be used to override assumptions if the developer file has a different structure. The expanded list of class properties to handle saved queries is shown below. The `QueryQ` is used to display a list of saved queries when the user selects the Load Query button.

```
QueryfileIDKey  &KEY
QueryFileID     ANY
QueryFileCategoryKey    &KEY
QueryfileCategory   ANY
QueryFileName   ANY
QueryFileQuery  ANY
QueryQ          &QueryQT
```

The `GetQueryFileInfo` method declares reference to a `TheFile` which is the file associated with the `FileManager` class, `AKey` which can be used to examine its keys, and `ThisGroup` which can be used to examine the fields in the file's record structure.

```
!------------------------------------------------------------------------------
PDSQLFilterCT.GetQueryFileInfo PROCEDURE()
!------------------------------------------------------------------------------
TheFile                         &FILE
AKey                            &KEY
ThisGroup                       &GROUP
  CODE
    IF SELF.QueryG.FM &= NULL THEN RETURN.
    DO r_GetKeysandFields

r_GetKeysAndFields  ROUTINE
  DATA
Keys    SIGNED
CategoryKey &KEY
  CODE
    TheFile &= SELF.QueryG.FM.File
    Keys=TheFile{PROP:Keys}
    ASSERT(NOT Keys < 2,'The Saved Query file needs at least two keys, one a primary id key, the other a category
    LOOP I# = 1 TO Keys
      AKey &= TheFile{PROP:Key, I#}
      CASE AKey{PROP:Components}
      OF 1
        IF SELF.QueryFileIDKey &= NULL
          ASSERT(AKey{PROP:Primary}=1,'Single component key is not primary.')
          ASSERT(AKey{PROP:Dup}<>1,'Primary key must not allowd duplicates.')
          SELF.QueryfileIDKey &= TheFile{PROP:Key, I#}
        END
        IF SELF.QueryFileID &= NULL
          SELF.QueryFileID &= SELF.QueryG.FM.GetField(AKey,1)
        END
      OF 2
        IF SELF.QueryFileCategoryKey &= NULL
          ASSERT(AKey{PROP:Primary}='','Single component key is not primary.')
          ASSERT(AKey{PROP:Dup}<>1,'Named query key must not allowd duplicates.')
          SELF.QueryFileCategoryKey &= TheFile{PROP:Key, I#}
        END
        IF SELF.QueryFileCategory &= NULL
          SELF.QueryfileCategory &=SELF.QueryG.FM.GetField(AKey,1)
        END
        IF SELF.QueryFileName &= NULL
          SELF.QueryFileName &= SELF.QueryG.FM.GetField(AKey,2)
        END
      ELSE
      !-- If there are keys that do not meet this spec, they you'll have to do some hand coding in the app embed.
      END
    END
    IF SELF.QueryFileQuery &= NULL
      ThisGroup &= TheFile{PROP:Record}
      SELF.QueryFileQuery &= WHAT(ThisGroup,eQueryPosition)
    END
```

The method gets the number of keys in the file and then looks for a primary key with a single ID field and one with two fields, one for the category and one for the name. It then assumes the query field itself is in a position `eQueryPostion`, an equate in the Translation file that can easily be changed by the developer. As series of assert statements, show below, checks whether these fields and assumptions hold true.

```
    !-- If you get an error on one of these, you need to hand code missing field
     !   references. Use the GetQueryFileInfo method before the parent call embed.
     !   Assign the following:
     !     SELF.QueryFileIDKey
     !     SELF.QueryFileID
     !     QueryFileCategoryKey
     !     QueryFileCategory
     !     QueryFileFileName
     !   Values can be set in the SetQueryFile method for loading.
     !   When Saving, values can be primed in the QueryFilePrimeUpdate method.
    ASSERT(NOT SELF.QueryFileName &= NULL,'Query field was not found. See notes above')
    ASSERT(NOT SELF.queryFileID &= NULL,'Primay field not found. See notes above')
    ASSERT(NOT SELF.QueryG.Category &= NULL,'Category not found. See notes above')
    ASSERT(NOT SELF.QueryFileQuery &= NULL,'The query field in the query file not found. See notes above')
```

If the file has a different structure then the developer will need to modify the class or use the virtual embed before the parent call to assign reference variables. If a field has been assigned, the method will not override it. As an example, this would accommodate having queries segregated by user or security level.

Now that the query `FileManager`, fields, and keys are known, it is possible to load the query queue with saved queries. To handle fields that may have been added and allow the developer to filter records if needed, two methods are added, one for priming key fields before setting the file, and one for validating records.

```
!------------------------------------------------------------------------------
PDSQLFilterCT.LoadQueryQ PROCEDURE()
!------------------------------------------------------------------------------
RV                                BYTE
```

```
  CODE
    IF SELF.QueryG.FM &= NULL
      RV=1
    ELSIF SELF.QueryG.FM.Open()
      RV=1
    ELSE
      SELF.QueryG.FM.UseFile()
      SELF.QueryG.FM.ClearKey(SELF.QueryFileCategoryKey)
      SELF.SetQueryFile()
      LOOP UNTIL SELF.QueryG.FM.Next()
        SELF.Debugout('Category '&SELF.QueryFileCategory,'Filter.loadQueryQ')
        IF SELF.QueryFileCategory<>SELF.QueryG.Category
          BREAK
        END
        CASE SELF.ValidateQueryRecord()
        OF 1 !Record Filtered
          CYCLE
        OF 2 !Out of Range
          BREAK
        END
        ASSERT(NOT SELF.QueryFileName&=NULL,'Query File name is NULL')
        ASSERT(NOT SELF.QueryQ &= NULL,'QueryQ is null')
        CLEAR(SELF.QueryQ)
        SELF.QueryQ.QueryName=SELF.QueryFileName
        SELF.QueryQ.QueryID=SELF.QueryFileID
        ADD(SELF.QueryQ,SELF.QueryQ.QueryName)
      END
      SELF.QueryG.FM.Close
    END
    RETURN RV
```

Without embed code, the `ValidateQueryRecord` method simply returns 0. The `SetQueryFile` sets the file for record processing.

```
 !---------------------------------------------------------------------------
 PDSQLFilterCT.SetQueryFile PROCEDURE()
 !---------------------------------------------------------------------------
   CODE
     ASSERT(NOT SELF.QueryG.Category &= NULL OR NOT SELF.QueryG.Category='','Query category must be specified')
     SELF.QueryFileCategory=SELF.QueryG.Category
     SET(SELF.QueryFileCategoryKey,SELF.QueryFileCategoryKey)
 LoadQueryAction
 !---------------------------------------------------------------------------
 PDSQLFilterCT.LoadQueryAction PROCEDURE()
 !---------------------------------------------------------------------------
   CODE
     SELF.Query=SELF.LoadQuery()
     IF ~CLIP(SELF.Query) THEN RETURN.
     DISPLAY
     IF SELF.CheckQuery()
       SELF.ExecQuery
     END
```

The `LoadQuery` method is called when the Load Query button is pressed. It calls the window procedure previously discussed. If a query is loaded, it checks it and then executes it if the check succeeds.

```
 !---------------------------------------------------------------------------
 PDSQLFilterCT.LoadQuery PROCEDURE()
 !---------------------------------------------------------------------------
   INCLUDE('PDSQLFILTER.TRN','LOADQUERY')
 RV                      STRING(10000)
   CODE
     RV=SELF.Query
     OPEN(LoadQueryW)
     GET(SELF.QueryQ,1)
     SELECT(?List,1)
     SELF.Translate(LoadQueryW)
     SELF.DebugOut('QueryQ Records: '&RECORDS(SELF.QueryQ),'Filter.LoadQuery')
     ACCEPT
       CASE EVENT()
       OF EVENT:Accepted
         CASE FIELD()
         OF ?DeleteBtn
           GET(SELF.QueryQ,CHOICE(?List))
           IF NOT ERRORCODE()
             SELF.QueryFileID=SELF.QueryQ.QueryID
             IF ~SELF.QueryG.FM.TryFetch(SELF.QueryFileIDKey)
               SELF.QueryQ.QueryID=SELF.QueryFileID
               IF ~SELF.QueryG.FM.DeleteRecord()
                 GET(SELF.QueryQ,SELF.QueryQ.QueryID)
                 IF NOT ERRORCODE()
                   DELETE(SELF.QueryQ)
                 END
               END
             END
           END
         OF ?SelectBtn
           GET(SELF.QueryQ,CHOICE(?List))
```

```
          IF NOT ERRORCODE()
            SELF.QueryFileID=SELF.QueryQ.QueryID
            IF ~SELF.QueryG.FM.TryFetch(SELF.QueryFileIDKey)
              RV= SELF.QueryFileQuery
              SELF.LastQueryID=SELF.QueryFileID
              BREAK
            END
          END
        END
      END
    END
    RETURN CLIP(RV)
```

The `LoadQuery` method, called `LoadQueryAction`, opens a window displaying a list of saved queries. These may be Selected, Deleted, or the window may be closed without a selection. The queue handling is quite simple. The `FileManager` does all the work of handling the file.

Note that there is an `INCLUDE` statement referencing a section in the Translation file. This section contains the window definition. The `Translate(WINDOW pWin)` method is called immediately before the `ACCEPT` loop after opening the window. This allows for multi-language development. You can modify the appearance of this window to suit your own standards and translate it to the language used in your application. You could also add a Help ID.

```
SECTION('LOADQUERY')
LoadQueryW WINDOW('Load Query'),AT(,,205,203),|
FONT('MS Sans Serif',8,,FONT:regular),CENTER,GRAY
       LIST,AT(8,8,145,180),USE(?List),VSCROLL,|
       FORMAT('480L(2)|_M~Saved Queries~@s120@#1#'),FROM(SELF.QueryQ)
       BUTTON('&Select'),AT(164,132,35,14),USE(?SelectBtn),DEFAULT
       BUTTON('&Delete'),AT(164,152,35,14),USE(?DeleteBTN)
       BUTTON('Cl&ose'),AT(164,176,35,14),LEFT,STD(STD:Close)
     END
```

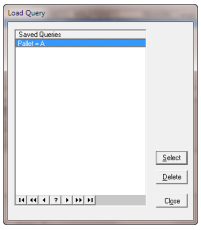

Figure 3. Load query window.

Like the `LoadQuery` method, `SaveQuery` has a window that is declared in the Translation file. It is called from the Save Query button which is enabled only if the `QueryBuilder` field contains a valid query.

```
!-----------------------------------------------------------------------------
PDSQLFilterCT.SaveQuery PROCEDURE()
!-----------------------------------------------------------------------------
Overwrite                BYTE
ThisName                 STRING(250)
SavName                  STRING(250)
  INCLUDE('PDSQLFILTER.TRN','SAVEQUERY')
  CODE
    OPEN(SaveQueryW)
    SELF.Translate(SaveQueryW)
    IF NOT SELF.LastQueryID &= NULL
      SELF.DebugOut('Last Query ID: '&SELF.LastQueryID,'Filter.SaveQuery')
      SELF.QueryFileID=SELF.lastQueryID
      IF ~SELF.QueryG.FM.TryFetch(SELF.QueryFileIDKey)
        SELF.DebugOut('Last Query ID Found','Filter.SaveQuery')
       ?LastQueryName{PROP:Text}=SELF.TranslateString(_Query_LastName)&|
        ': '&SELF.QueryFileName
        SavName=SELF.QueryFileName
        ThisName=SELf.QueryFileName
        Overwrite=TRUE
        UNHIDE(?Overwrite)
        UNHIDE(?LastQueryName)
        DISPLAY
      ELSE
        SELF.DebugOut('Last Query ID NOT Found','Filter.SaveQuery')
        SELF.LastQueryID &= NULL
      END
    END
```

If the user previously loaded a query using the Load Query button, the `LastQueryID` property will have been set (will not be null) and the last query loaded. The user is then also given an option to overwrite the last query or save the current query as a new one. The name of the last query is displayed in a string on the window along with a title that is translated.

```
SECTION('SAVEQUERY')
SaveQueryW WINDOW('Save Query'),AT(,,279,73),FONT('MS Sans Serif',8,,FONT:regular),CENTER,GRAY
       PROMPT('Query &Name:'),AT(14,8),USE(?Prompt1)
       ENTRY(@s120),AT(66,8,193,10),USE(ThisName)
```

```
        CHECK('Overwrite last loaded query?'),AT(66,20),USE(OverWrite),HIDE
        STRING('***'),AT(66,32),USE(?LastQueryName),HIDE
        BUTTON('&OK'),AT(97,52,41,14),USE(?OkButton),DEFAULT
        BUTTON('&Cancel'),AT(141,52,41,14),USE(?CancelButton),STD(STD:Close)
      END
_Query_LastName  EQUATE('Last Name')
```

The window contains hidden fields that are displayed only if a query was previously loaded.
The section also contains an equate for a string the developer can modify.

```
    ACCEPT
      CASE ACCEPTED()
      OF ?OKButton
        IF Overwrite
          SELF.QueryFileName=ThisName
          SELF.QueryFileQuery=SELF.Query
          SELF.QueryFilePrimeUpdate()
          IF NOT SELF.QueryG.FM.Update()
            SELF.QueryQ.QueryName=SavName
            GET(SELf.QueryQ,SELF.QueryQ.QueryName)
            IF NOT ERRORCODE()
              SELF.QueryQ.QueryName=ThisName
              PUT(SELF.QueryQ)
              SORT(SELF.QueryQ,SELF.QueryQ.QueryName)
            ELSE
              SELF.DebugOut('Error Getting Prior Queue '&CLIP(SELF.QueryQ.QueryName)&' '&ERROR(),'FILTER.SaveQuery
            END
          ELSE
            CYCLE
          END
        ELSE
          SELF.QueryFileName=ThisName
          SELF.QueryFileQuery=SELF.Query
          SELF.QueryFileCategory=SELf.QueryG.Category
          SELF.QueryFilePrimeInsert()
          IF NOT SELF.QueryG.FM.Insert()
            SELF.QueryQ.QueryName = ThisName
            SELF.QueryQ.QueryID = SELF.QueryFileID
            ADD(SELF.QueryQ,SELF.QueryQ.QueryName)
          ELSE
            CYCLE
          END
        END
        SELF.LastQueryID &= NULL
        POST(EVENT:CloseWindow)
      END
    END
```

The method also calls two place holder methods that the developer can user to assign other fields on an update or insert: `QueryFilePrimeUpdate` and `QueryFilePrimeInsert`.
The Figure 4 shows the method in action.

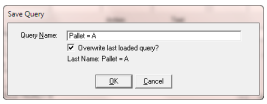

Figure 4. Save query window.

Other methods

The reset method clears the filter, clears the `Alpha` field and the `LastQueryID`, applies the filter, and resets the browse using the browse class methods.

```
!-----------------------------------------------------------------------------
PDSQLFilterCT.Reset PROCEDURE()
!-----------------------------------------------------------------------------
  CODE
    SELF.Query=''
    SELF.SetSQLFilter()
    IF ~SELF.GetFieldFEQ(eFld:Alpha)
      CHANGE(SELF.FieldQ.FieldFEQ,'')
    END
    SELF.LastQueryID &= NULL
    DISPLAY()
    SELF.Bc.ApplyFilter()
    IF SELF.CheckQuery()
      SELF.BC.ResetFromBuffer
      SELF.BC.PostNewSelection
    END
```

The `BuildText` method copies the contents of the text entry to the `QueryBuilder` field and then clears the text field.

```
!-----------------------------------------------------------------------------
PDSQLFilterCT.BuildText PROCEDURE()
!-----------------------------------------------------------------------------
  CODE
    IF SELF.GetFieldFEQ(eFld:Alpha) THEN RETURN.
    IF CONTENTS(SELF.FieldQ.FieldFEQ) = '' THEN RETURN.
    SELF.BuildAction(CLIP(CONTENTS(SELF.FieldQ.FieldFEQ)))
```

```
    CHANGE(SELF.FieldQ.fieldFEQ,'')
    DISPLAY
    SELF.CheckQuery()
```

The `BuildField` method copies the current selection in the `QueryFieldQ` to the `QueryBuilder` field. The procedure example did this with every new selection. This could be duplicated by calling `BuildField` from the `TakeNewSelection` method which is otherwise an empty shell virtual procedure.

```
!-----------------------------------------------------------------------------
PDSQLFilterCT.BuildField PROCEDURE()
!-----------------------------------------------------------------------------
  CODE
    IF SELF.GetFieldFEQ(eFld:QueryField) THEN RETURN.
    GET(SELF.QueryFieldQ,CHOICE(SELF.FieldQ.FieldFEQ))
    SELF.BuildAction(SELF.QueryFieldQ.QueryField)
    DISPLAY
    SELF.CheckQuery()
```

Concluding thoughts

At this point the class is working with separate class INC, CLW, TRN, and TPL files. It is worth cleaning up and organizing the files according a bit. Copying revisions back into the PD Class Generator template can be helpful in doing this, but is certainly not necessary.

It can, for example, be helpful to list the class methods in some order. I usually list the `Construct/Destruct` methods first, followed by private methods, and then all other methods alphabetically. Using the Class Generator, they can easily be put in order in both the INC and CLW files.

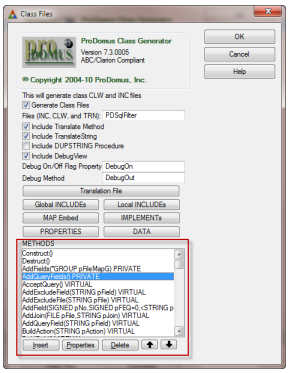

Figure 5. An ordered list of methods.

Class data - global and local includes

Data and typed definitions, such as queues and groups, can go in into the CLW file event though they may be referenced as parameters or property definitions in the INC file. If an item needs to be referenced in the application outside of the classes, then they need to go into the INC file. In this example, all the typed queues are in the CLW file. The `SavedQueryGT` is declared in the INC file because an instance of it needs to be declared in the procedure so it can be passed as a parameter to the Init method.

If the class is to implement something, then the file declaring the implementation needs to be in the INC file. An example in this case is the ABWINDOW.INC file because the `WindowComponent` is implemented is implemented by the class. The ABBROWSE.INC file can be in the CLW file.

Translation file

Translation files should contain all the items that can be modified by developer. They are designed to be copied to the application directory so they will not be overwritten by new installs. Third party providers sometimes change these files, so they should be checked when a new install is done.

Class files with sections can easily be included in many locations as was done here.

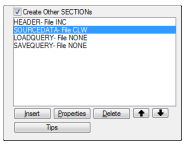

Figure 6. Translation (TRN) file sections and locations.

Overview

Creating shell class templates that instantiate your class object, INC, CLW, and Translation (TRN) is a useful first step in developing a new class. From there you can hand code and test in incremental steps, reworking names and code as needed. The PD Class Generator is a good tool for doing this, but it can and has been will done by taking a sent of shell files and replacing about a dozen interrelated labels across the set of files.

A working procedure is a good starting point, but often presents tough challenges which changing from procedure specific code to more generic class code. As you develop procedures, it is important to think about places where other developers might want to modify your methods and provide virtual methods that appear in the embed list for them to

do this. If your class is going to be used by other developers for applications in other languages or in an international market, it is important to provide for customization and multi-language translation. A method for debugging can be quite handy. Fortunately, Clarion, with the way it handles classes and the many properties available, offers a great environment for doing this.

If you like the idea of the SQL filter class, but are concerned about the amount of real estate it takes up, you might consider a derived class that populates a load and reset button and maybe a single line `QueryBuilder` field that could be the target of the Browse drag and drop. Other controls could be put on a class window. At first blush, this seems like it would be easy and a nice challenge to test your class writing skills. If you succeed, send Dave Harms an article with code.

Writing this class and template was a interesting challenge keeping me busy over the holidays. I was concerned about working with SQL in Clarion, but this turned into a relatively small issue. Discovering that `PROP:Alias` let me use the file names in queries was an important early find. Unexpectedly, probably the greatest challenge was getting view, file, and field properties, as the label, field, and fields properties and the entities to which they can be applied and their indexing are difficult to sort out. The PDClass Generator was extremely helpful in the first stage of creating the set of coordinated and ready-to-compile shell files.

One final caveat – the specific SQL filter control template appears to be working but has not been tested with a large database or more than a very simple view and application. It also not been user tested, so there may be issues. If you plan to use it, do some more testing.

Download the source

Philip S. Will is President of ProDomus, Inc, a SoftVelocity Third Party Accessories Partner and Clarion applications developer. He has been a presenter at several Clarion conferences, and has published articles on Internationalization and template writing. His principal third party products include PD Lookups, PD Translator Plus, and PD 1-Touch Date Tools. Philip has been coding in Clarion since 1991.

Article comments

BACK TO TOP

Tip of the Week #9: Code Completion Window Width

By Dave Harms

Posted February 2 2011

Code completion is an enormously popular feature in modern programming IDEs; I've quickly become as dependent on it in C7 as I am in Visual Studio.

Not everyone is a fan of having the editor's help when it comes to writing code. Some argue that code completion (or Intellisense, to use Microsoft's name for this feature) makes developers more dependent on the IDE and less dependent on their own memories.

That's probably true. And ideally we'd all be so brilliant and have such retentive minds that we could write pages of code without referring to documentation.

But apps and the APIs they call are getting so big that most of us benefit from a little help.

C7's code completion can be a little wonky at times (when calling class methods I often find myself backing up and pressing "." a second time) but I've already become quite attached to it. And a few days ago I came across a little feature that makes me like it even more.

I was looking through the IDE options when I noticed something under Text Editor | Clarion for Windows (Figure 1).

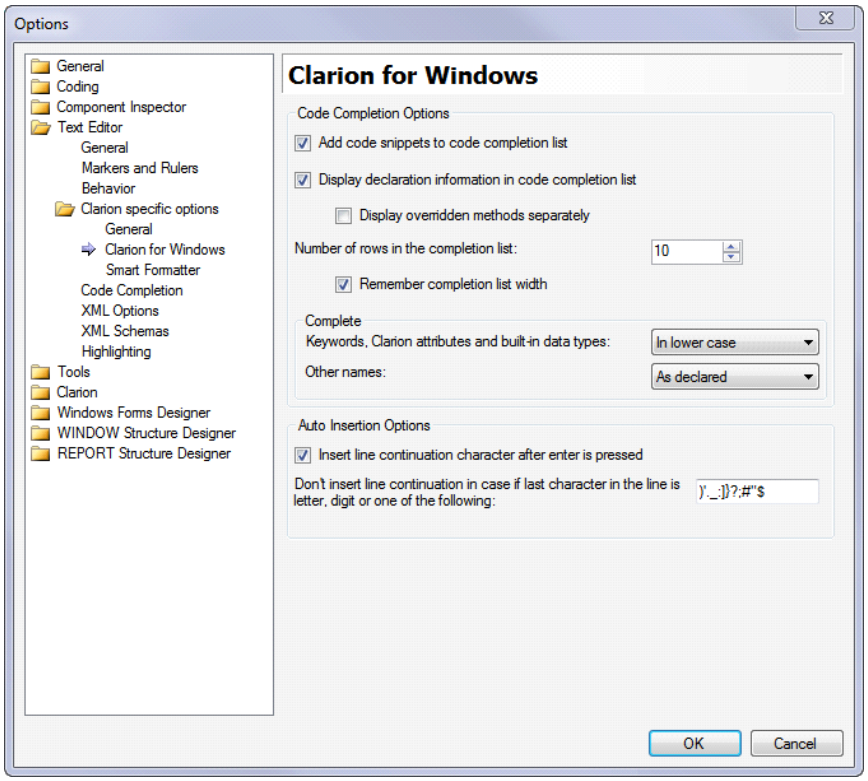

Figure 1. The Clarion text editor options

For the first time I noticed a checkbox for "Remember completion list width".

It had never occurred to me that I could change the code completion popup window's width. So I gave it a try. Figure 2 shows the code completion list; in Figure 3 I've dragged the window so it's quite a lot wider.

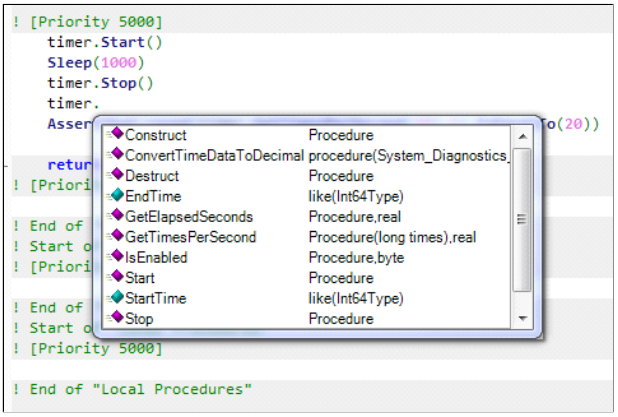

Figure 2. Narrow list

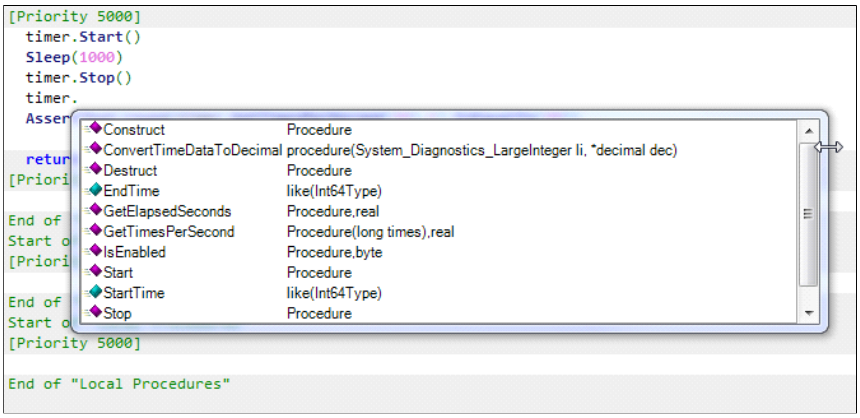

Figure 3. Wide list

I'm not convinced that the checkbox does anything, because on my machine the IDE seems to remember the last width I set whether the option is checked or not. But that doesn't matter a whole lot - I just like being able to choose the width of the window.

Now if I could only adjust the width of the first column....

David Harms is an independent software developer and the editor and publisher of Clarion Magazine. He is also co-author with Ross Santos of Developing Clarion for Windows Applications, published by SAMS (1995), and has written or co-written several Java books. David is a member of the American Society of Journalists and Authors (ASJA).

Article comments

BACK TO TOP

Webinar this Friday: If you can't test your code, how do you know it isn't crap?

By Dave Harms

Posted February 8 2011

Most Clarion developers use Clarion because of the AppGen; they let it do the grunt work, and they slot their own code into various embed points.

But it's possible to overuse embeds. Put all your source code there and you can't really reuse it, short of copying and pasting.

So what's the answer?

The first step is to start pulling that embed code out into reusable procedures or, even better, classes.

But that's just the first step.

The next, and absolutely vital, step is to create automated tests you can execute against that code.

Most of us don't really test our code. We write some embed code, and then we run the app and we click some buttons and type some text and check to see if everything's running as expected. Or worse, we let our customers do the testing.

That's little more than throwing code up against the wall to see what sticks. It's not systematic, it's not easily repeatable, and above all it isn't automated.

If you want your applications to be competitive, they have to be reliable. And the first step to ensuring the reliability of your entire application is to begin verifying the reliability of the code you write.

The webinar

On Friday, February 11 2011 join me in a ClarionLive webinar on unit testing. During the webinar I will take the ClarionLive Class Bash ASCII file reading/writing classes and submit them to unit testing, using the ClarionTest framework. I'll show how to write tests for the

existing classes and I'll explore a number of refactorings to make those classes more testable and reusable.

The webinar starts at 9 a.m. PST (GMT-8). Please register ahead of time.

David Harms is an independent software developer and the editor and publisher of Clarion Magazine. He is also co-author with Ross Santos of Developing Clarion for Windows Applications, published by SAMS (1995), and has written or co-written several Java books. David is a member of the American Society of Journalists and Authors (ASJA).

Article comments

by Andrew Barnham on February 8 2011 (comment link)

Shame I am in such an ungodly timezone. Will watch once vid is online.

Clarion language syntactically has a powerful advantage for providing supporting tools for a comprehensive test framework: the ACCEPT loop and a restricted number of GUI widgets. On my TODO list, I will hopefully have lots of free time in a month or two, is to hook a test framework into Clarion2Java. The runtime/compiler itself has a test suite of over 2000+ test cases; but not framework yet to write tests for clarion code itself; and having done unit testing for years now I am very eager to throw some test code around the apps I look after.

Automated testing of GUI apps, even webapps to some extent, is painful. Simulating KB and mouse gestures is highly brittle process (as evidenced in test cases in clarion2java runtime). But with ACCEPT loop you only need to record event list + a little bit of logic to track implied behaviour/state of some of the controls. i.e. ACCEPT on entry control changes value of the variable the ENTRY USE() wraps. The beauty of it is that on your test runner playback you don't need to startup an actual GUI environment. You could have a test running ACCEPT loop that is GUI-less and whose primary purpose is to test behaviour/logic of the application. The test obviously doesn't provide 100% coverage, but the resulting test code will be simple and easy to manage, non-brittle and exercises the truly important bits: a good trade off in my mind.

Also the work effort should not be substantial; a few weeks only. Part of that would involve writing a recorder; like recorder in Selenium. The recorder will monitor your app as you manually use it, automatically write the clarion (or java) test code and you only need tweak it to iron out non obvious bits, such test conditions that may change: like todays date etc.

---

by Dave Harms on February 8 2011 (comment link)

Andrew,

Those are some very interesting suggestions.

I won't actualy be dealing with the GUI at all - I'll be focusing on testing the business

logic. I'll also be touching on some of the differences between test-first and test-after development.

Dave

BACK TO TOP

Tip of the Week #10: Speed Up Source Navigation With Bookmarks

By Dave Harms

Posted February 11 2011

Clarion 7 has a couple of nice navigation aids that can really speed up your coding, one of which is the Bookmarks pad.

Bookmarks are useful both inside and outside of the AppGen, but they're most helpful when you're working with source files.

Open a source file, put the cursor on a line where you want the bookmark, and press Ctrl-F2. Two things will happen. First, you'll see a bookmark icon appear next to the line of text, and second, on the bookmarks pad you'll see the same line of text and the source file name (Figure 1).

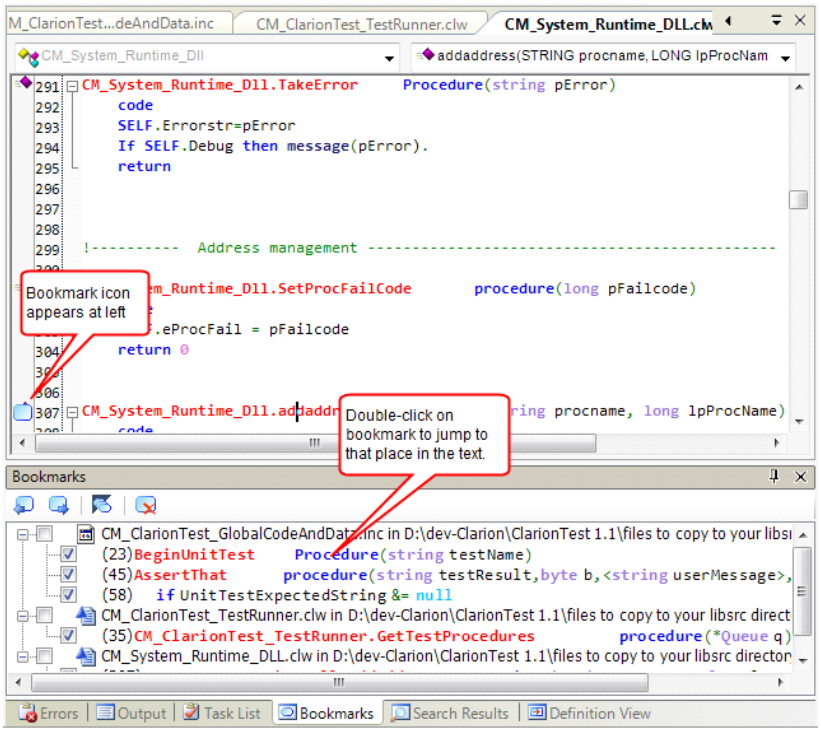

Figure 1. Adding a bookmark

Obviously you want to select some meaningful text for your bookmark - if you press Ctrl-F2 you'll get a blank line in the Bookmarks pad. You'll still see the name of the file containing the bookmark, and the line number, but the whole point of bookmarks is to help you navigate source code. If there isn't any descriptive code where you want to leave the bookmark, add a comment and bookmark that.

Bookmarks float with the line they're on, so if you insert some text above, the bookmark moves down. You can use F2 to move to the next bookmark in the file, or Alt-F2 to move to the previous bookmark. To go to a bookmark in another document you need to actually click on that bookmark.

Saving bookmarks

Bookmarks are saved between sessions, sometimes. It depends.

Bookmarks in non-generated source files that are part of a project will still be there if you restart Clarion and reopen the project/solution. So for a hand-coded project, bookmarks work great.

If you add a bookmark to some *generated* code, and leave Clarion and come back, that bookmark will be there. But if you regenerate the file (and only changed files are actually regenerated) any bookmarks in that file will be lost.

If you have source files in a solution folder, that is a folder that you create under the solution, bookmarks in those files are *not* saved. You'll have to add these files to your project if you want to preserve bookmarks.

This last situation describes how I do almost all my class development: I set up a solution folder for the class files and open them from there. I've posted a feature request (PTSS 37658) to preserve bookmarks for these files as well.

Bookmarks do work in the embeditor, but they're even more transient than they are in solution folder source files. As soon as you close the embeditor any bookmarks you've created in the embeditor go away. It probably shouldn't be too surprising that these bookmarks vanish, however, since the AppGen actually generates the displayed source each time you use the embeditor.

On the other hand you can close a source file with a bookmark and the bookmark remains. Double-click on the bookmark and the IDE will open the file and display the bookmarked position.

You can also use the Bookmarks pad to navigate between bookmarks inside the embeditor and bookmarks in source files.

The next time you find yourself shuffling back and forth between different points in source files or within the embeditor, take a moment to drop in a couple of bookmarks.

David Harms is an independent software developer and the editor and publisher of Clarion Magazine. He is also co-author with Ross Santos of

Developing Clarion for Windows Applications, published by SAMS (1995), and has written or co-written several Java books. David is a member of the American Society of Journalists and Authors (ASJA).

Article comments

by William Tetley on February 17 2011 (comment link)

Dave,

Another great tip that made me slap my forehead and say "I had no idea". Keep them coming sir.

by Dave Harms on February 18 2011 (comment link)

Thanks Tony, I'm glad you're finding them useful!

BACK TO TOP

MagGem: Validating Credit Card Numbers

Posted February 16 2011

Credit card validation is one of those things that you don't pay much attention to until you need to do it. I'm not talking about processing payments - most of us use services for that - but about that first check that verifies the card number itself is valid, as a guard against incorrectly entered data.

There is a standard methodology for card number validation: it's called the Luhn algorithm. It's basically a checksum formula, and is used for credit card numbers, US National Provider Identifier numbers and Canadian Social Insurance numbers.

Back in 2004 Abe Jimenez needed to do that kind of validation, and to his surprise he couldn't find a Clarion version of the code. So he wrote one.

Read the article now

Watch the MagGem in ClarionLive Webinar #94

Article comments

BACK TO TOP

MagGem: Backdoors and Other Tricks

Posted February 17 2011

Carl Barnes is one of the most technical Clarion developers I know. His knowledge of Clarion is encyclopedic; his code is precise. He's written quite a few articles for Clarion Magazine over the years, including Using KEYSTATE For Backdoors And Other Tricks. From Carl's introduction:

> The Clarion KEYSTATE function returns the status of the "shift type" keys (Shift, Ctrl, Alt), the lock keys (Caps Lock, Num Lock, Scroll Lock), and the Insert key (overwrite or insert). These keys are different from the rest of the keys on the keyboard in that they don't return a KEYCODE value to Clarion when pressed. It might sound like KEYSTATE isn't good for much more than displaying information on the status bar, but in fact this function is a great tool for detecting unusual keystroke combinations (including when the numeric keypad has been used), which you can use to implement hidden features in your applications. In this article I'll walk you through detecting key states, and I'll give some examples of useful hidden behaviors.

Carl goes on to explain in detail how you can use `KeyState()` and `Band()` to enable various non-obvious key combinations for things like security back doors.

Read the article now

Watch the MagGem in ClarionLive Webinar #95

Article comments

BACK TO TOP

Tip of the Week #11: The Task List

By Dave Harms

Posted February 18 2011

Last week's tip was about using bookmarks to navigate around source files. But as I pointed out in that article, there are some limitations to bookmarks. They don't persist in the embeditor, and they don't persist if the file you're editing isn't part of a project.

But there's another way to track specific locations in files that works with any open file and with the embeditor, and that's the task list.

Figure 1 shows the IDE options window with the General | Task List entry highlighted.

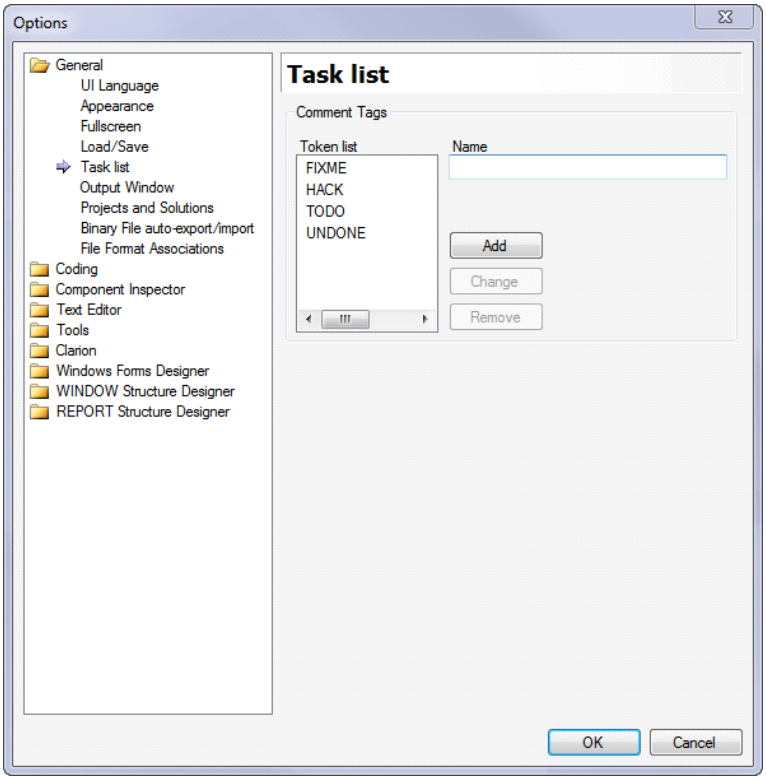

Figure 1. Task list options

The items in the task list are completely arbitrary - they're magic strings, if you like. Magic because, if you put them in text as a comment, the IDE will see that text as a bookmark. In

Figure 2 I've added a couple of !TODO comments and a !HACK comment. These task list items show up in the Task List pad.

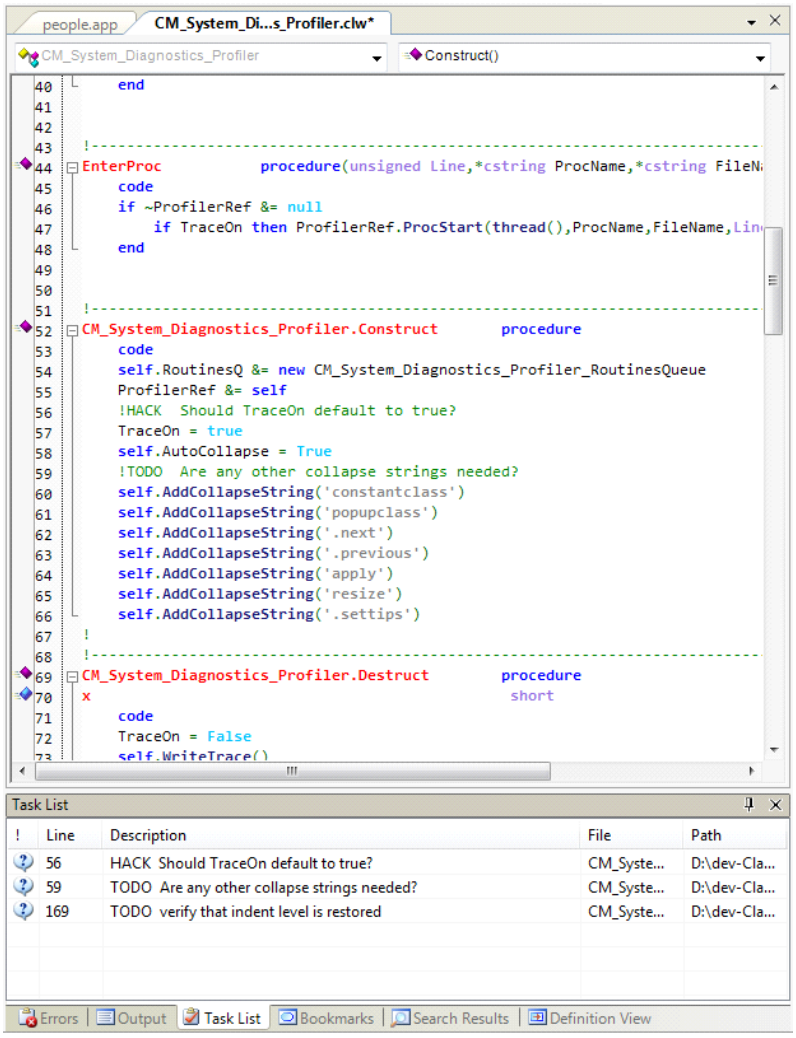

Figure 2. Adding task list items

You can add your own task list items, as Figure 1 indicates. For bookmarks you could use ! BOOKMARK or just !BKMRK. Or use any other text you like. Just enter that text preceded by an exclamation mark, followed by some comment text.

Task list items are persisted in the embeditor. In Figure 3 I've created a !BKMRK task list item and have added a corresponding comment.

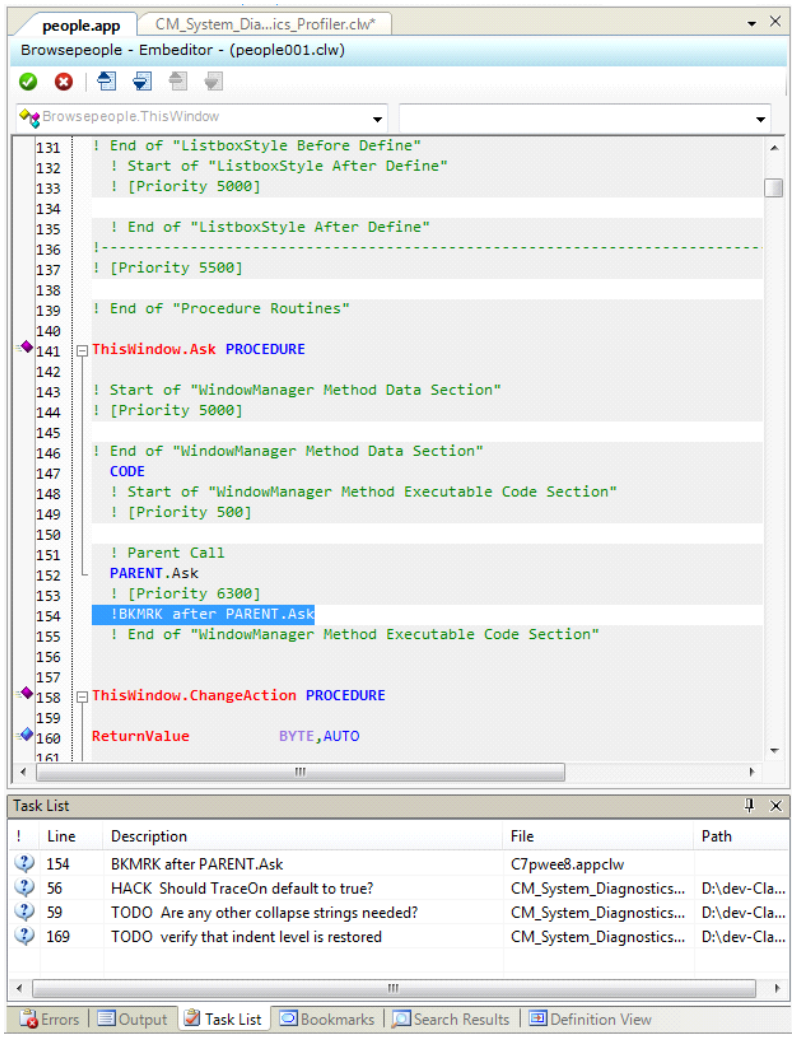

Figure 3. Adding a task list item in the embeditor.

There are two wrinkles to using task list items with the AppGen. One is that after you generate and come back into the embeditor, you'll see the task list item twice, once for the automatically generated embeditor temp file, and once for the permanent generated source file (Figure 4).

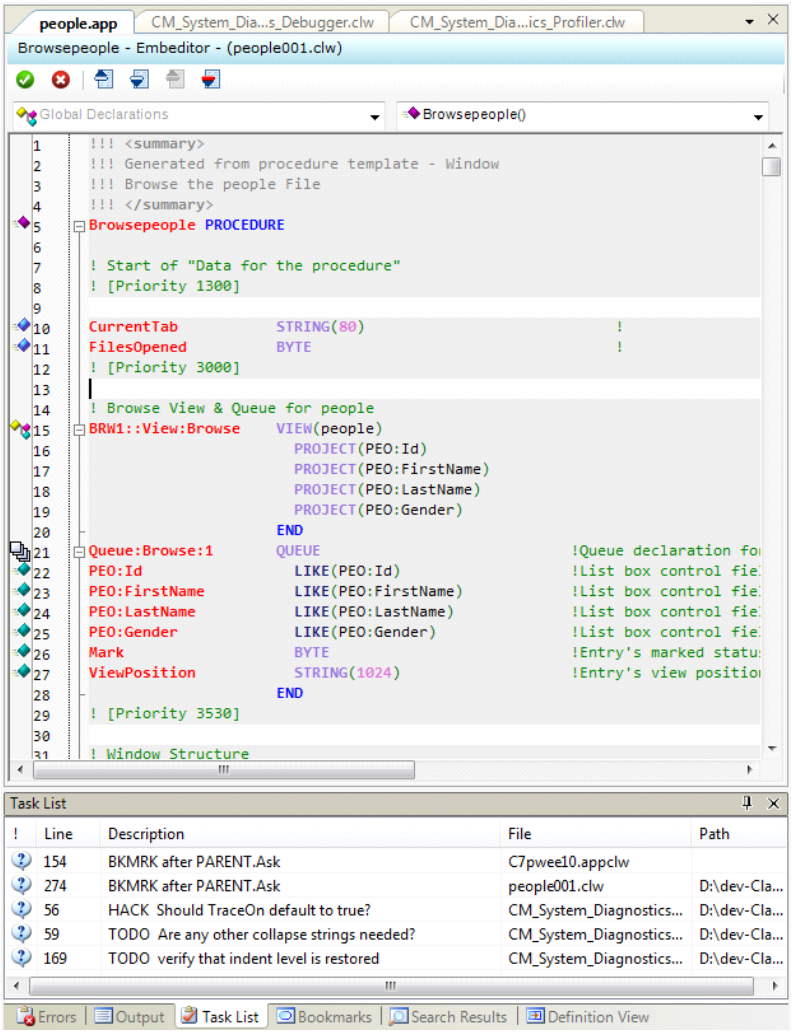

Figure 4. Duplicate task list items

Double-clicking on the temp file task takes you to the appropriate point in the embeditor. Clicking on the generated source file task takes you to that line in the generated source file.

The second problem with the task list is that whenever you generate code, the task list is cleared. If you have task list items in source files you have to close and reopen those files. I've reported this as PTSS 37689.

Until the task clearing bug is fixed there are really just two main cases for using bookmarks. One is within the embeditor, especially when you find yourself jumping around between different embed locations. The other is when you're doing hand code only and not generating apps.

Even with these restrictions, the task list can be a useful tool.

David Harms is an independent software developer and the editor and publisher of Clarion Magazine. He is also co-author with Ross Santos of Developing Clarion for Windows Applications, published by SAMS (1995), and has written or co-written several Java books. David is a member of the American Society of Journalists and Authors (ASJA).

Article comments

by Mark Riffey on February 18 2011 (comment link)

This. Is. AWESOME.

by Dave Harms on February 22 2011 (comment link)

Cool - glad you like it, Mark!

BACK TO TOP

Automagical Browse Refreshing

By John Morter

Posted February 18 2011

Automatic browse refreshing perhaps isn't groundbreaking for many in the Clarion community, but it is one of my favourite nifty tips that make developing apps with Clarion just so much easier (aka, automagical).

Automatic BrowseBox refreshing may be required when some event has occurred that the standard template logic is not naturally aware of. Some such examples are:

1. Making a CheckBox selection that is supposed to limit the contents of the BrowseBox
2. Making a FileDrop/Combo selection that is supposed to act as a filter for the BrowseBox
3. Taking some action on an entry/row in the BrowseBox, other than via one of the Update buttons, where you need the subsequent update to be reflected in the BrowseBox.

In Figure 1, the purpose of the FileDrop and CheckBox selections is to change the contents of the BrowseBox; The FileDrop selection is supposed to limit the List to only those records in the table related to a specified Publisher, and the CheckBox selection is supposed to override the Publisher selection to list all Brandings, regardless of Publisher.

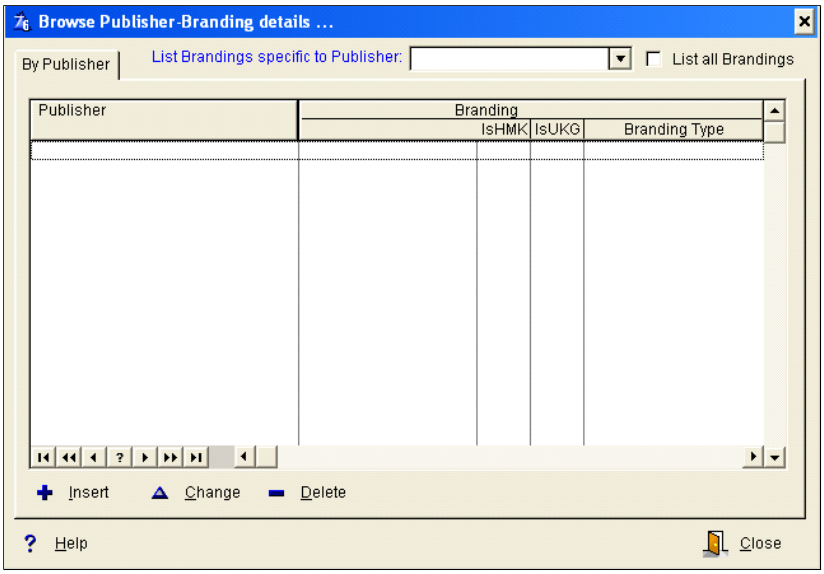

Figure 1. BrowseBox with non-standard influencing "events"

This being an SQL-based application, I'm applying these selections via a Filter on the browse, which will be converted by the ABC templates (version 6.73, in my case, but that's another story altogether!) into an `SQL WHERE` clause for server-side `SELECT`ion.

Figure 2. Filter expression to apply CheckBox & FileDrop selections

`ListALL` is a local variable that's set to `True` or `False` depending on whether the CheckBox is ticked (or "checked", in USA-speak) or not.

`PubRID` is another local variable that receives the record ID (RID) of the Publisher (Pub) that was selected via the FileDrop.

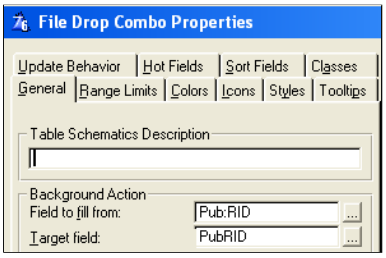

Figure 3. Action specified for when FileDrop selection completes

If the application is compiled at this point then everything works fine, *except* … the BrowseBox is not refreshed whenever the FileDrop or CheckBox selections change.

Consulting the ABC Library Reference results in the following suggestion;

> **Tip:** Use the `ResetSort` method followed by `UpdateWindow` to refresh and redisplay your ABC BrowseBoxes. Or, use the `WindowManager.Reset` method.

This may sound straightforward, but there are a couple of catches involved.

Firstly, there's the requirement to get the syntax right, and if you want an efficient result you'll need to choose the right embed-point(s) in which to call `ResetSort+UpdateWindow`.

Secondly, the `ResetSort+UpdateWindow` method is a sledgehammer approach.

As a dip back into the ABC Library Reference reveals: "The Reset method calls the ResetSort and UpdateWindow methods for *each* BrowseClass object registered by the AddItem method [and it] calls the ResetQueue method for *each* FileDropClass object registered by the AddItem method." (emphasis added).

This gets the job done, but it means that *all* BrowseBoxes and *all* drop lists on the window are updated … not just and only the one that is the subject of your request to be updated.

Now, this may not be a problem because very often there is only one BrowseBox on the

window, such as in my simple example. So what you get is what you wanted … assuming you've placed your calls to `ResetSort+UpdateWindow` in the correct embed-point(s).

However, there are also times when you have more than one BrowseBox on the window. And refreshing all BrowseBoxes, when you intended to update one of them, is wasteful of resources … *especially* when you're working with SQL tables.

You can avoid these inefficiencies by calling *`BrowseClassName`*`.Reset` instead, because that will limit the refresh to only the BrowseBox that belongs to the *`BrowseClassName`* … but there are still the syntax quirks and embed-point placement issues to contend with.

Fortunately, there is a better way *and* a much easier (this being the part I like best) way to force a refresh on a BrowseBox.

Before I reveal the punch-line though, yet another check of the ABC Library Reference for the `BrowseClass` reveals that; "The AddResetField method specifies a field that the browse object monitors for changes, then, when the contents of the field changes, refreshes the browse list."

What this means is that I can simply use the *`BrowseClassName`*`.AddResetField` method to declare a field that will be monitored for any change (of that field), and when any change is detected then the related BrowseBox will be refreshed. Brilliant !

Initially, it may seem that there are much the same syntactical and embed-selection challenges in applying this method as there is with using *`BrowseClassName`*`.Reset` … and you'd be right. However, the ABC Templates can handle all this automagically. There's a [Reset Fields] option on the BrowseBox template. Just add the appropriate field(s) to the list.

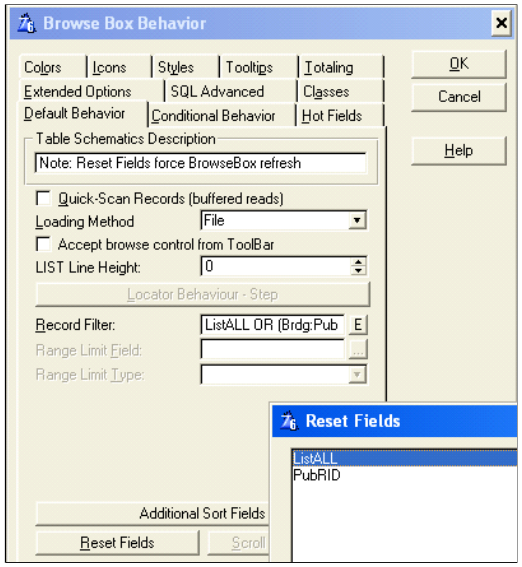

Figure 4. Specifying Reset "triggers" to force BrowseBox refresh

That's all I need to do: the ABC Templates will call the

BrowseClassName.AddResetField method for me … using its “knowledge” of where calls to this method should be placed, with the correct syntax, and with reference to the same *BrowseClassName* that’s managing the specific BrowseBox that I want to be refreshed.

Thereby, whenever any change to either of these variables is detected, the BrowseBox will be refreshed in the most efficient way possible … because monitoring for these changes is handled in standard BrowseClass and WindowManager logic (rather than depending on me to put some syntactically correct code into the right embed-point).

This same approach can be used any other time I need the BrowseBox to be refreshed, such as when some other action has changed the content of row(s) in the List.

My typical approach, in this case, is to use a simple BYTE variable as the Reset Field, and to “bump” the value of this variable whenever that “other action” has been, well, actioned. (eg. ResetTrigger += 1). This works fine, even if actioned more than 255 times (where 255 is the maximum value of a BYTE), because, for a BYTE variable, 255+1=0.

Nifty, eh ?

John Morter is Asia Pacific IT Manager for a brand-name multi-national and he's supposed to leave all the fun technical stuff for others to do. As a result, his Clarion work is developed under the nom-de-keyboard Flat Chat Solutions, where "flat chat" is an Australian expression meaning doing something at top speed / high velocity.

Article comments

by Lisa Daugherty on February 21 2011 (comment link)

This is a great article... and nicely written. I've always done it the BrowseClassName.Reset way, so I can't wait to try this new easier method on my next browse. Thanks!

by John Morter on February 22 2011 (comment link)

Thanks for the kind words, Lisa.

It's a funny thing, isn't it ... how we use a tool without necessarily using all its features/capabilities - as per my use of Excel. One day I noticed the "Reset" button and thought; "I wonder what that's for" (?!).

BACK TO TOP

MagGem: Capitalize The Right Way

Posted February 22 2011

As Mike Hanson pointed out a couple of years ago in Formatting Names Using Proper Case, Clarion's text capitalization is rudimentary at best. The CAP attribute arbitrarily forces the first letter of each word to uppercase, which isn't always what you want. Type john smith and CAP gives you John Smith, but john smith md becomes John Smith Md, von richtoven becomes Von Richtoven (the 'von' should remain lowercase), and mcdowell becomes Mcdowell rather than McDowell.

While there are some rules that can be applied to this problem, there are also many special cases. Happily, Mike has written a class that applies both rules and exceptions to the problem of capitalization, so if his code doesn't cover some situation you can easily rectify that with a method call or two.

Read the article now

Watch the MagGem in ClarionLive Webinar #97

Article comments

BACK TO TOP

MagGem: Everything You Ever Wanted To Know About Strings

Posted February 24 2011

Strings. Business applications are loaded with them. You could say that we spend a good chunk of our time as software developers simply dealing with strings: creating them, copying them, moving them, evaluating them, retrieving and storing them.

A lot of the time we're using a lot more string data than is necessary. That's because most of the time we're not passing strings to procedures, we're passing copies if strings. That's not a big deal if the calls are few or the strings are small, but even in the age of the desktop supercomputer copying too much string data can slow down your app and cause excessive memory usage.

Back in 2003 Jim Gambon wrote String Flinging, a two part series on the basics of string handling in Clarion. Among other things Jim covers string references, string slicing, String vs CString, casting, and returning strings.

This is a must read for anyone wanting to get the most out of their string handling.

Read the article now

Watch the MagGem in ClarionLive Webinar #96

Article comments

BACK TO TOP

Tip of the Week #12: Searching/Finding Files Using Redirection

By Dave Harms

Posted February 27 2011

Many of the recent Tips of the Week have focused on C7. This week's tip is a feature that's actually more fully implemented in C6.

You probably know all about Clarion's redirection (.RED) files. These are used by the IDE to locate files required by your project. Without a redirection file you'd have to have all of your application's source and other required files in one big directory. That would be a real pain, particularly since those files would include everything that's currently in your libsrc directory, like the ABC classes.

Redirection files are also used to determine where you generate files, which is a good subject for a future tip. But this week's tip is about using redirection to find the files *you* are interested in.

On Clarion 6's File menu there's an option called Search Files (Figure 1).

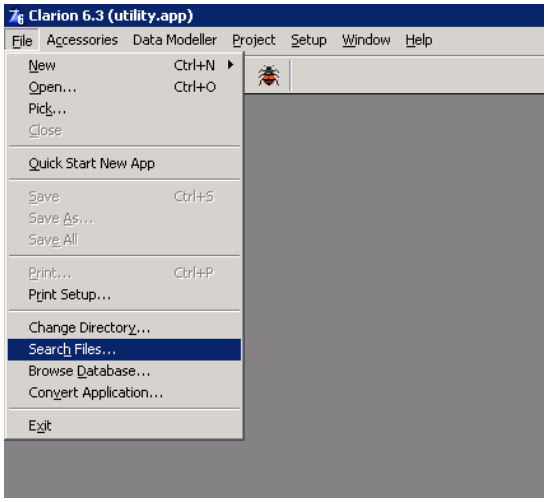

Figure 1. Search Files

Select this option and you'll see the File Search dialog (Figure 2).

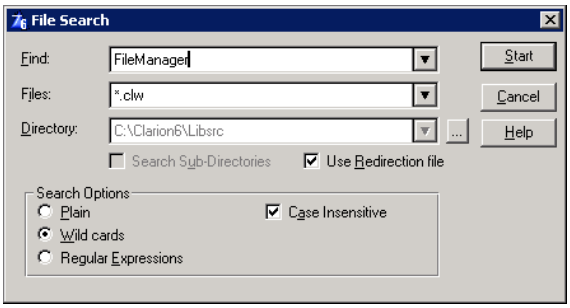

Figure 2. The File Search dialog

Note that in Figure 2 I've selected the Use Redirection file checkbox. That disables the Directory field, so rather than search a specific directory (which is useful it itself) Clarion will search through all of the directories specified in the redirection file. You also have the option of using either regular expressions or wild cards in the search.

There are a few limitations in the search. Clarion 6 will list all of the files that match your search term, but will only take you to the first instance in each file. Also the regular expression support seems a bit doubtful. Searching for ^FileManager, where the ^ means the beginning of the line, doesn't work. Actually start and end of line regex characters don't in C7 either (unlike, say, Visual Studio).

But File Search is still a very handy tool in Clarion 6. And the IDE keeps the last search results on hand, so you don't have to redo the search if you close the results window.

Clarion 7 has several features that loosely correspond to the Clarion 6 functionality. As Figure 3 shows, there are some different text search options, including the current document, the current selection, all open documents, the project, the solution, and a folder. Searching across multiple documents is still a bit flaky, in my experience. When it works, it's great.

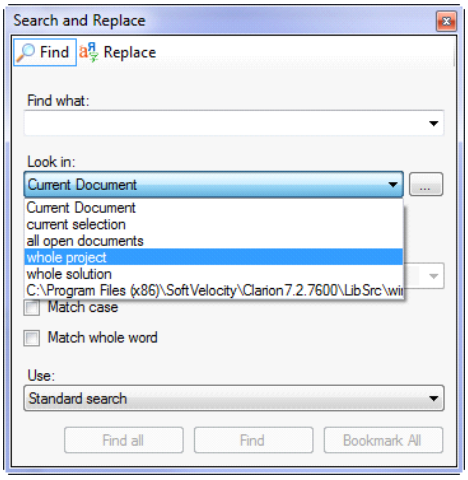

Figure 3. Clarion 7 search options

If you specify a folder (Figure 4) you can also specify the file extensions to search. And in

both cases you can do standard, wildcard and regular expression searches.

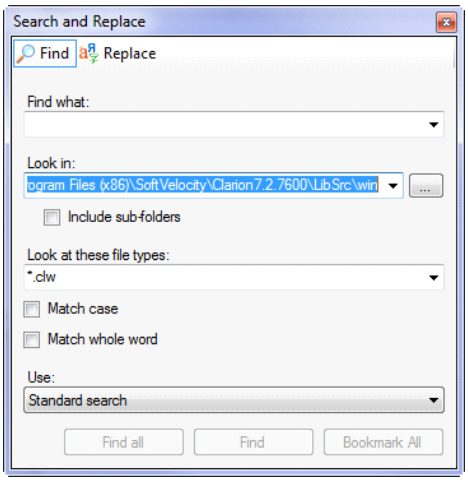

Figure 4. File search options

There's also a way to open files via redirection via the File menu (Figure 5).

Figure 5. Opening a file using redirection

This is a loss of functionality in that, to my knowledge, you can't search via the redirection file in Clarion 7. Also when you open a file using redirection you may not know where that file comes from. Happily there's a solution to that problem. Just hover your mouse over the file's tab (Figure 6).

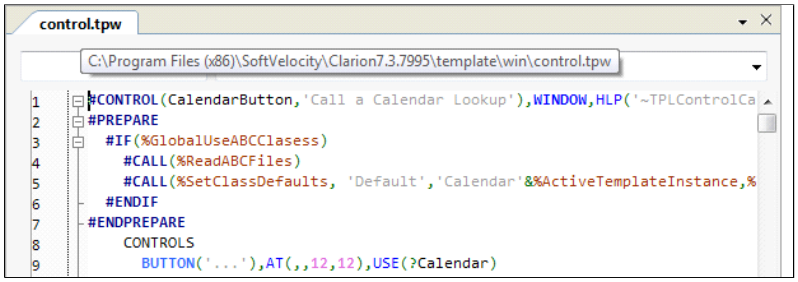

Figure 6. Checking a file's path

You can also right-click on the tab and either copy the path to the clipboard or open the file's folder.

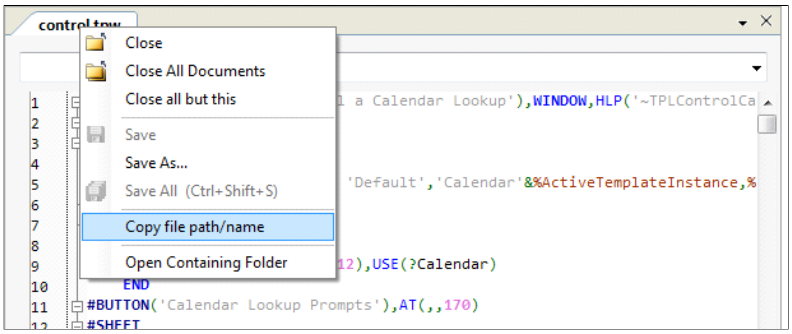

Figure 7. File tab options

There are some important differences between Clarion 6 and Clarion 7 when it comes to searching and opening files via redirection. Clarion 7 is actually better in some respects, but I still wish I had the ability to search via the redirection file.

David Harms is an independent software developer and the editor and publisher of Clarion Magazine. He is also co-author with Ross Santos of Developing Clarion for Windows Applications, published by SAMS (1995), and has written or co-written several Java books. David is a member of the American Society of Journalists and Authors (ASJA).

Article comments

BACK TO TOP

Highlighting Text With RTF The Easy Way, Part 1

By Dave Harms

Posted February 28 2011

If you've attended a ClarionLive webinar recently (and if you haven't, you should) you'll know that John and Arnold asked me to do a weekly special called MagGems, in which I talk about an article from a past issue of ClarionMag. Recently I mentioned Steve Bottomley's article Enriching The User's Experience With RTF Displays. Steve's idea was to use the Clarion RTF control to display color highlighted text to the user, something that isn't possible with a regular text control.

I remember thinking at the time that I really liked Steve's approach. But it wasn't until just recently that I needed similar functionality for my own work.

Like a great many of Clarion examples, especially from the early years, the original RTF code is embed code. There's nothing wrong with embed code per se, except that it's stuck in an embed. And because it's in an embed you can't easily test the code or reuse the code.

There's a saying that to a man with a hammer, everything looks like a nail. I'm discovering that to a developer with unit testing tools, everything looks like an opportunity for test-driven development.

In this article I'll walk you through my approach to turning Steve's RTF idea into a class using that test-driven development process.

The tools

I do Test-Driven Development (TDD) using ClarionTest, a unit testing framework I originally wrote based loosely on some of the experiences I've had with unit testing in .NET. Recently John Hickey and I have made a number of improvements to ClarionTest, and I'll be writing more about those changes in the near future. For right now, if you want to follow along go to the Google Code repository and download the latest version of ClarionTest. There are two zips, one of just source and the other complete with the ClarionTest executable. You'll also find several files you need to copy to your libsrc directory as well as a template to copy to your template directory (and you need to register the template). Once you've done all that you're ready to go.

Getting started

I've written about test-driven development before, so I won't go into the rationale in any

detail here. But as with all TDD, the starting point is the test. And that means doing a little bit of setup first.

The first thing I need to do is create a DLL that will contain my unit test procedures. And a DLL needs a name. I generally create one test DLL for each class I create, and I gave the DLL the same name as the class followed by _Tests. So before I do anything (and this is *only* because of my naming convention) I need to know the name of the class I haven't yet created.

A class naming digression

Up until recently I didn't have any kind of strategy for naming my Clarion classes. I tended to keep the names fairly short, and that's fine when you only have a few classes to worry about. But once you start accumulating dozens of classes, or hundreds, short, cryptic file and class names are a hindrance.

I've taken a cue from .NET namespaces. Namespaces are a bit like prefixes. In Clarion, two files can have fields with the same name, and the names don't conflict because each file has its own prefix. But namespaces are more than a way to avoid naming conflicts - they provide for a hierarchy of prefixes that you can use to organize your classes by their functionality or area of concern.

I really wish Clarion Win32 had namespaces, but it doesn't so I have to fake them. I thought about using names with colons (like `System:String`), but I prefer to have class names that exactly match the filenames, and colons aren't legal in filenames. After discussing this with several other developers I settled on an underscore as a separator. And I further decided to prefix all the ClarionMag classes with `CM_`.

A few of the classes I've created under this scheme are:

```
CM_System_Diagnostics_Debugger
CM_System_Diagnostics_Profiler
CM_System_IO_File
CM_System_IO_FileInfo
CM_System_String
CM_Text_Address
```

These may seem like very long class names, and they're definitely clunkier than namespaces, but Clarion 7's code completion pretty much takes the sting out.

Choosing namespaces is a tricky business, and I've already come to regret some of my choices. But as I already have a `CM_Text_Address` class, I'm going to go ahead and call this class `CM_Text_RTF`. So that will make my test dll `CM_Text_RTF_Tests`.

In Figure 1 I'm creating my apps in the directory of the same name, and I have Auto create project subdir turned off.

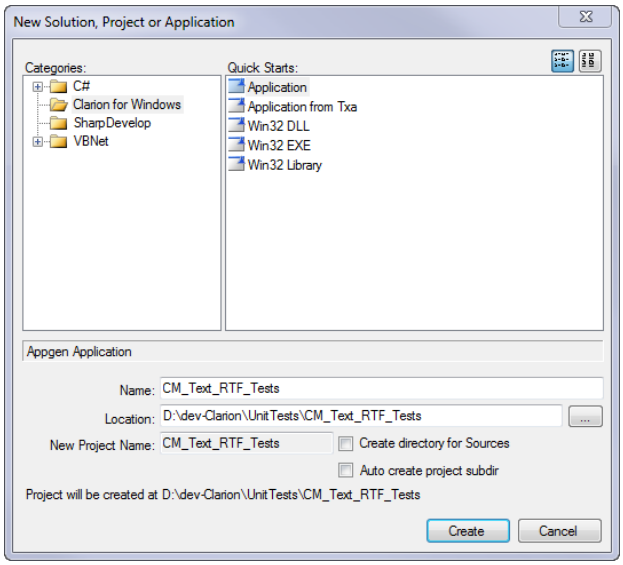

Figure 1. Creating the test DLL, step 1

I'll create this as a DLL (Figure 2) but in fact I'll need to update the project data or C7 will try to compile it as an EXE.

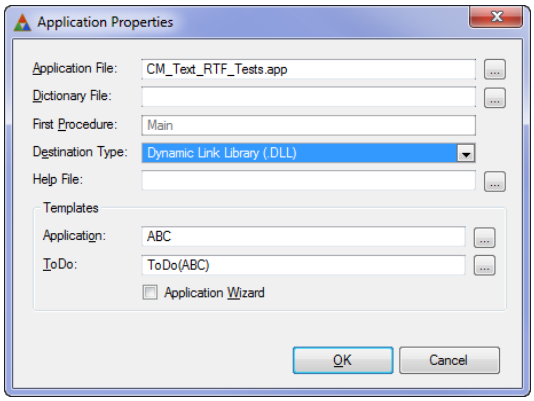

Figure 2. Creating the test DLL, step 2

In Figure 3 I've opened the project's properties from the solution explorer and have set the target to DLL instead of EXE.

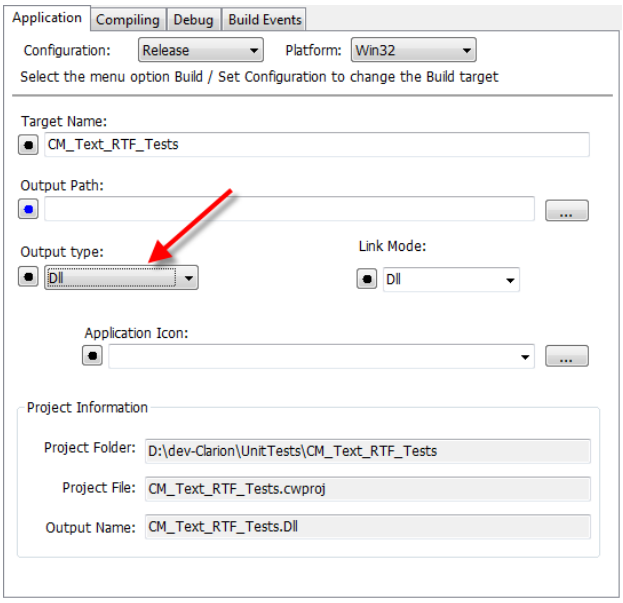

Figure 3. Setting the target to DLL.

While I'm in the project data I might as well set the post-build event. I want the ClarionTest application to automatically load this DLL and execute any tests. I always have ClarionTest.exe in my UnitTests directory, above the directories for the individual test DLLs so it's readily accessible. The `/run` parameter tells ClarionTest to run the tests immediately.

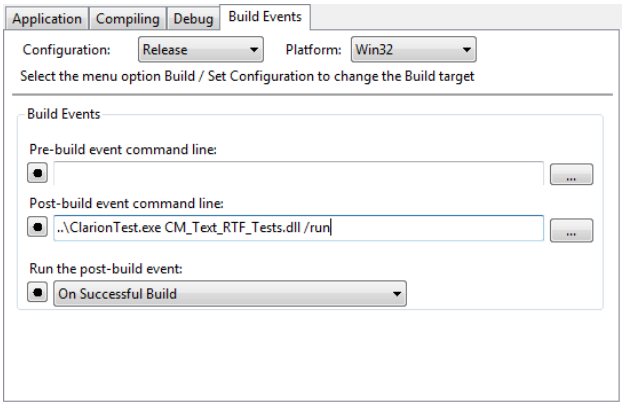

Figure 4. Setting the post-build task

I'll need to configure this DLL as a test DLL recognizable by ClarionTest. To do that I go to the global extensions tab and add the ClarionTest TestSupportIncludes extension (Figure 5).

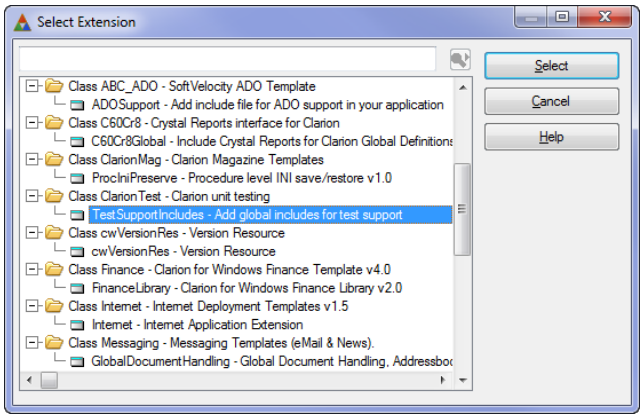

Figure 5. Enabling ClarionTest support

That may seem like a lot of work to prepare the test DLL, but it's really not that bad. It could be made even easier by a utility template. There's one more item for the Todo list.

I'm ready to create my first test procedure. Now, what do I want to test?

The requirement

My requirement is simple: I want to be able to take some plain text, convert it to RTF, and specify which words in that text should be shown with a specified foreground and/or background color. I'll call my first test `AddText_HighlightWord_VerifyRTF`.

In Figure 6 I've chosen the Test Procedure default (note that I'm on the Defaults tab).

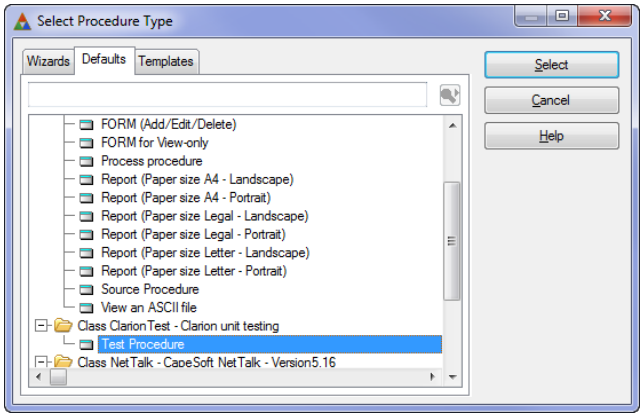

Figure 6. Choosing the test procedure default

The test doesn't do anything yet, but I'll go ahead and make it to make sure everything's fine. After the make completes the post-build tasks loads up ClarionTest and runs my test procedure (Figure 7).

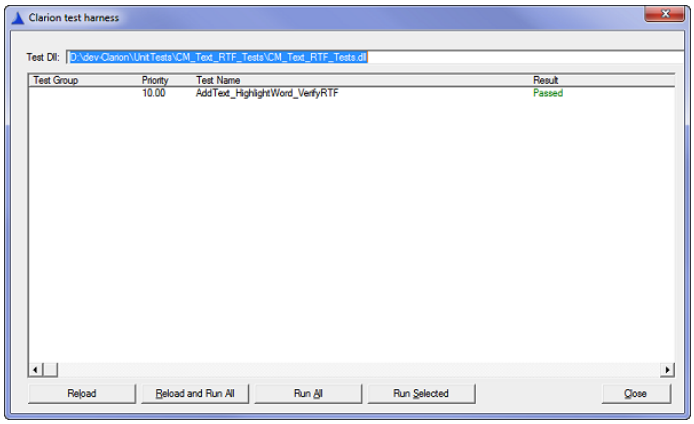

Figure 7. Running the test

Now it's time to add some code, which will break the test.

Starting with broken code

In test-driven development (TDD) you write the test first, even before you write the code. Not only will that break the test, but the code isn't even going to compile at first. That seems a bit counter-intuitive, but in fact it can be a powerful tool because it forces you to think about how you want to *use* your code before you think about how to *write* your code.

I try to make my initial test code (well, all my test code) as expressive as possible. I want to be able to read that code and know instantly what it's doing. So if I have an RTF object called, oh, rtf, I might write this code:

```
rtf.SetText('The word red should be displayed in red.')
rtf.HighlightText('red',color:red)
```

That seems pretty straightforward. But I'm not really testing anything - I need to verify that the RTF class contains the right text so I can place it in an RTF control. I'll need something like:

```
AssertThat(rtf.GetText(),IsEqualTo('dunno what should go here yet'))
```

`AssertThat` and `IsEqualTo` are utility methods provided by the ClarionTest framework. When you load up a DLL in ClarionTest and run the tests, the ClarionTest window will display either a Pass message for that test or the results of the first AssertThat that failed.

Something to test against

The bit that's still missing from my unit test is what the expected text should look actually like. I can generate RTF with the Clarion RTFNotepad example application. In Figure 8 I've created the text I want.

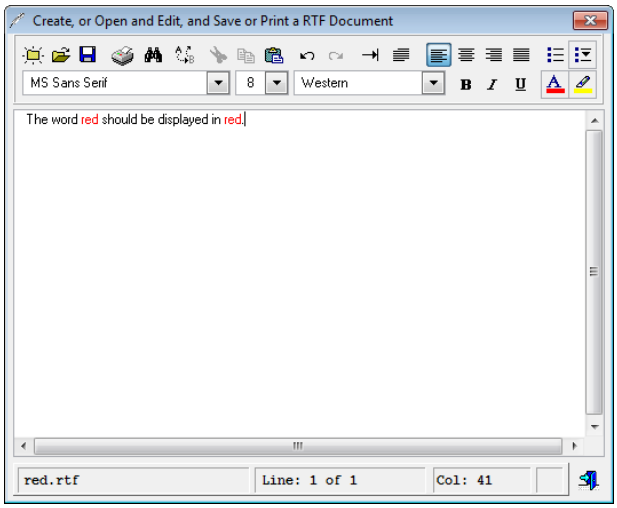

Figure 8. Creating the RTF text

If I save that text to a file and open the file with notepad I see the following (some line breaks added):

```
{\rtf1\ansi\ansicpg1252\deff0\deflang4105
{\fonttbl{\f0\fnil\fcharset0 MS Sans Serif;}}
{\colortbl ;\red0\green0\blue0;\red255\green0\blue0;\red8\green0\blue0;}
{\*\generator Msftedit 5.41.21.2509;}\viewkind4\uc1\pard\cf1
\highlight0\f0\fs17
The word \cf2 red \cf1 should be displayed in \cf2 red\cf1 .\cf3\par
}
```

That may look like gibberish, but it's really not too difficult to follow (although if you want entertainment, try saving that text as a Word RTF doc).

Among other things, RTF files contain groups of characters defined by braces. Another way to view the document is like this:

```
{\rtf1\ansi\ansicpg1252\deff0\deflang4105
      {\fonttbl
        {\f0\fnil\fcharset0 MS Sans Serif;}
    }
    {\colortbl ;\red0\green0\blue0;\red255\green0\blue0;\red8\green0\blue0;}
    {\*\generator Msftedit 5.41.21.2509;}
    \viewkind4\uc1\pard\cf1
    \highlight0\f0\fs17
    The word \cf2 red \cf1 should be displayed in \cf2 red\cf1 .\cf3\par
}
```

You can see that the document begins with an `\rtf` keyword followed by `1`, which is the version number. The character set is ANSI code page 1252, followed by default font and language indicators. The default font is font 0 which is defined in the `\fonttbl` section.

The critical feature for this RTF class is the \colortbl section:

```
{\colortbl ;\red0\green0\blue0;\red255\green0\blue0;\red8\green0\blue0;}
```

The color table contains one or more RGB color values, which correspond to color indexes starting with 0. Color table entries are followed by a semicolon, and can be omitted in which case they are assumed to be the default color. In the above example the first color is omitted. If red, green and blue are all zero the color is black. If all are 255 the color is white.

The `\*\generator` group is, I suspect, unnecessary, but I'll include it anyway.

After the last of the groups comes the document area:

```
\viewkind4\uc1\pard\cf1
\highlight0\f0\fs17
The word \cf2 red \cf1 should be displayed in \cf2 red\cf1 .\cf3\par
```

The `\viewkind` keyword indicates the view mode, and 4 means normal. Other views are none, page layout, outline, master document and online layout.

The `\uc` keyword indicates the number of bytes used for Unicode characters. That's not going to be an issue for my usage, but perhaps it would be for other languages.

The `\pard` keyword resets the paragraph properties so they are not inherited from the previous paragraph. The `\cf1` keywords sets the character foreground color to index 1. That's followed by `\highlight0` to set text highlight to color 0, which I don't believe has any affect in this example because `\highlight` is not used in text itself. Then comes the font number, `\f0` (referring to the font table) and the font size in half points (`\fs17` = 8.5 points).

The displayed text begins at the first non-keyword value, and thereafter keywords affect the display of the following text. So `\cf2` preceding the word "red" means set the foreground color to color table entry 2 (remember this is zero based and the first entry is omitted), followed by a reset to color table entry 1. Following the text "red" there's a color switch back to color table entry 1, which has a value of black.

The `\par` keyword at the end really isn't needed as it indicates a new paragraph, but there is no following text.

Beginning to code

It's time to take a first stab at some code. Figure 1 shows the code as added to the test procedure in the embeditor.

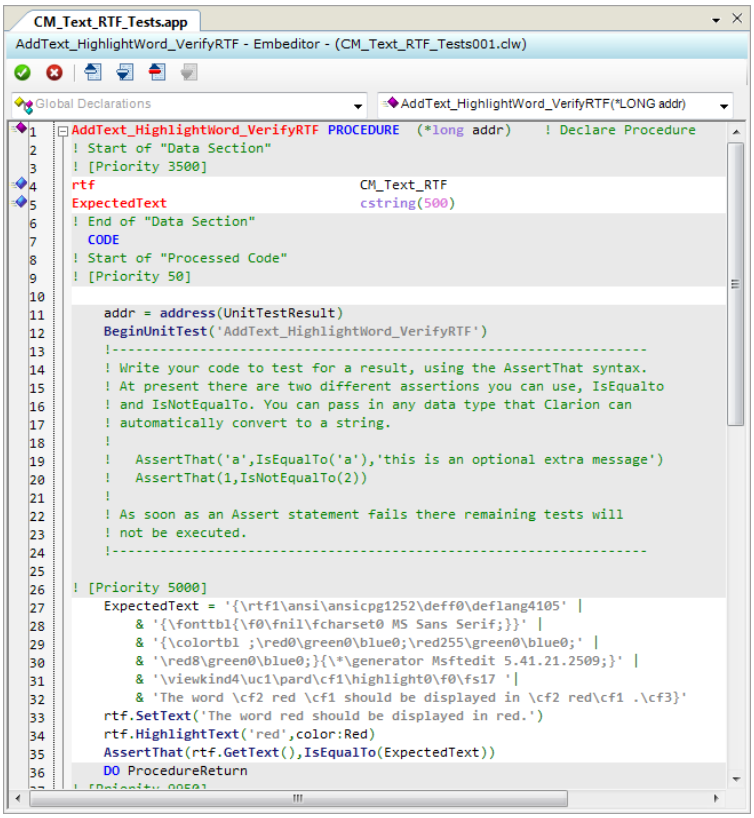

Figure 9. The initial test code

Just for fun I press compile, and of course the compile fails dismally.

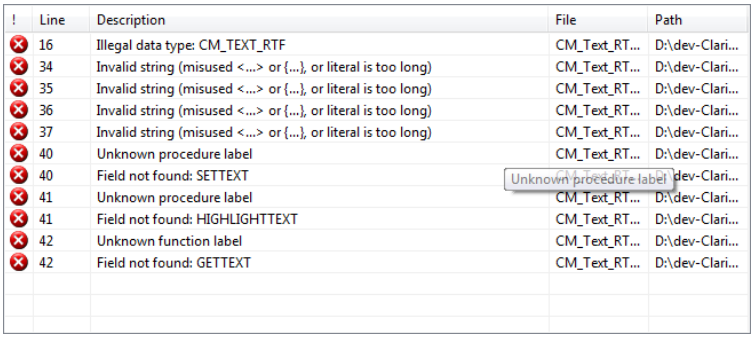

Figure 10. Compile failure

Writing the class

The first thing I need to do is to get a successful compile, and to do that I need to write a class with some stub methods. I create two files, CM_Text_RTF.inc and CM_Text_RTF.clw. And I create a new solution folder called Classes and add these two files so I'll have ready access to them while I'm working on my tests.

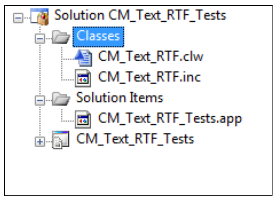

Figure 11. Adding the class files to the solution

CM_Text_RTF.inc initially looks like this:

```
    OMIT('_EndOfInclude_',_CM_Text_RTF_)
_CM_Text_RTF_   EQUATE(1)

CM_Text_RTF     CLASS,TYPE,MODULE('CM_Text_RTF.clw'),LINK('CM_Text_RTF.clw')
GetText           procedure,STRING
HighlightText     procedure(string text,long foregroundColor)
SetText           procedure(string text)
                  End

    _EndOfInclude_
```

The `Omit` statement is simply insurance against someone forgetting to use `,Once` on the Include statement for this file.

CM_Text_RTF.clw initially looks like this:

```
                    MEMBER


                    MAP
                    END


    INCLUDE('CM_Text_RTF.inc')


CM_Text_RTF.GetText             procedure !,STRING
    code
    return ''

CM_Text_RTF.HighlightText       procedure(string text,long foregroundColor)
    CODE

CM_Text_RTF.SetText             procedure(string text)
    code
```

The class methods are just stubs.

I also add this line to the application's After Global Includes embed:

```
include('CM_Text_RTF.inc'),once
```

On compiling I get the errors shown in Figure 12, not on the class but on my test code.

!	Line	Description	File	Path
⊗	34	Invalid string (misused <...> or {...}, or literal is too long)	CM_Text_RT...	D:\dev-Clari...
⊗	35	Invalid string (misused <...> or {...}, or literal is too long)	CM_Text_RT...	D:\dev-Clari...
⊗	36	Invalid string (misused <...> or {...}, or literal is too long)	CM_Text_RT...	D:\dev-Clari...
⊗	37	Invalid string (misused <...> or {...}, or literal is too long)	CM_Text_RT...	D:\dev-Clari...

Figure 12. Compile errors

I forgot to double up my braces, which have special meaning in Clarion strings and must be escaped. After replacing `{` with `{{` (only the opening brace matters) I get a successful compile. But predictably the test fails (Figure 13). After all, `GetText()` is just returning an empty string.

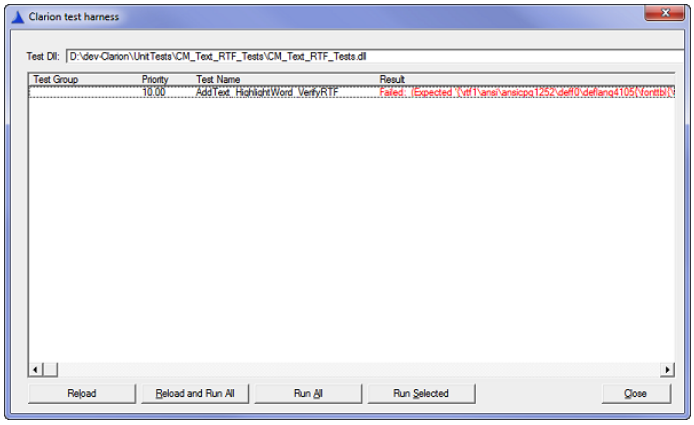

Figure 13. The failed test.

With a compiling, but failed test in hand I'm ready to start some serious coding. In the second instalment I'll walk through that process and show the finished class.

Download the source

Download ClarionTest

David Harms is an independent software developer and the editor and publisher of Clarion Magazine. He is also co-author with Ross Santos of Developing Clarion for Windows Applications, published by SAMS (1995), and has written or co-written several Java books. David is a member of the American Society of Journalists and Authors (ASJA).

Article comments

BACK TO TOP