Running Legacy COBOL Programs by Proxy With COBOL.NET

John D. Haney
john.haney@nau.edu
College of Business Administration
Northern Arizona University
Flagstaff, AZ 86011-5066

ABSTRACT

Microsoft's .NET Integrated Development Environment (IDE) provides a process where old legacy COBOL programs can appear as if they were written in a contemporary language such as C#. This is accomplished within the .NET environment by creating a solution that consists of two projects. The first contains the C# program with the graphical user interface. The second contains the legacy program and a COBOL proxy program that provides the link between the C# program and the legacy program. A COBOL data object class supplies the mechanism for the transference of data between the proxy program and the C# program. By the use of this process only slight modifications to the legacy program are necessary to run a legacy COBOL program by proxy. This project would most appropriately be integrated in an advanced programming course.

Keywords: COBOL, .NET, C#, legacy, proxy.

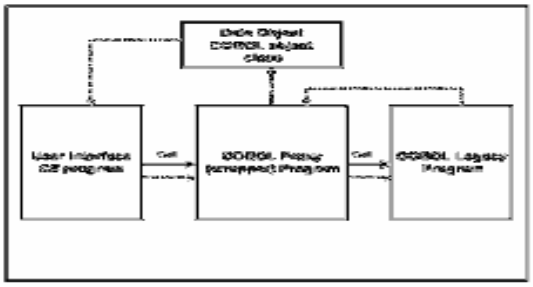
INTRODUCTION

Since its inception, COBOL has been a major part of the development of business information systems (Malek, 2002), and has adjusted as changes in programming methodology have arisen. Most noteworthy was the development of top-down design as a reaction to 'spaghetti' code. The addition of graphical user interfaces (GUIs) changed the complexion of programming, and the world of COBOL responded in two ways. First, GUIs were added to COBOL compilers, such as MicroFocus's Dialog system. Another response was to interface legacy COBOL systems with GUI oriented languages like Microsoft's Visual Basic. The trend toward object-oriented programming has also entered the world of COBOL (Malek, 2002). The initial reaction was to place object-orientation into the COBOL compilers (Price, 1997). However, given the flexibility of Microsoft's .NET platform, it is only natural and logical that there would be an inclusion of COBOL into the .NET environment (Narayana, 2001).

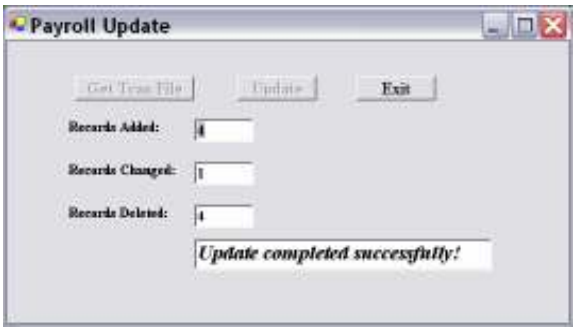
Part of Microsoft's philosophy for the .NET Framework is that programmers should be able to use the programming language best suited to their application (Malek, 2002). While C# and Visual Basic are the primary programming languages within the .NET environment, other languages are available. COBOL is one of those languages. Within the .NET Framework applications can be developed as stand-alone Windows applications or as web-based applications. The focus of this study is on a Windows-based application using MicroFocus's Net Express with .NET. The application, written in C#, controls the execution of a legacy COBOL program. The result is the execution of an old legacy COBOL program with a contemporary user interface, as if the program had been written in an object-oriented language such as VISUAL BASIC or C#. The heart of this structure is a proxy class that can easily be expanded to work with any existing COBOL program, hence 'running legacy COBOL programs by proxy.'

OVERVIEW OF SYSTEM

The process of running legacy COBOL programs within the .NET environment entails three required components and one optional component. The most obvious required element is the legacy COBOL program. Another is a program written in the .NET environment, such as a Visual Basic or C# program. In this example a C# program is used. These two must be tied together, which is accomplished with a COBOL program that works as a link between the C# program and the legacy COBOL program. An optional component is a COBOL object class which provides for storing and retrieving data between other components. The relationship of these elements is shown in the following diagram.

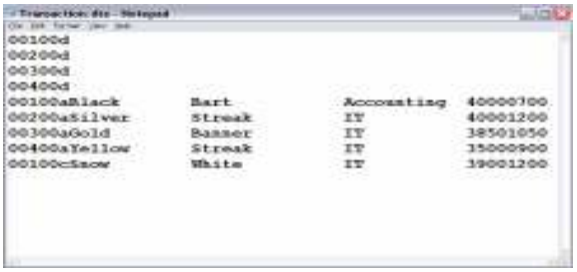


The functionality of the legacy COBOL program is to update a payroll master file. A transaction file drives a batch update that adds, changes, and deletes records. The update statistics: records added, changed, and deleted along with a completion message are sent back to the C# program. The mechanics of the system are fairly straightforward. The C# program, which provides the user interface, does two processes. First, the transaction data file is located using a common dialog box, and the path and file name are placed into a string data member. The second process, triggered by clicking the Update button, calls the COBOL proxy program. This sends the transaction file name, as an argument, and returns the update statistics into an instance of the data object. After control is retuned back to the C# program the statistics are then placed into the textboxes.

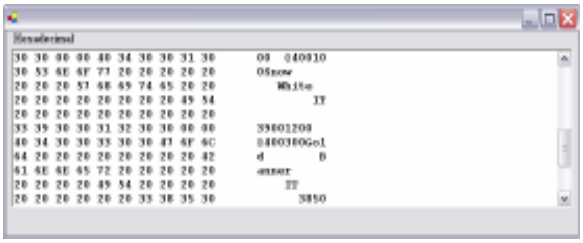


The COBOL proxy program receives the transaction file name and places it into its linkage section. Then the proxy program calls the COBOL legacy program, sending the transaction file name and receiving back the update statistics: records added, changed, deleted, and a completion message. These statistics are then placed into the data object, which is returned back to the C# program.

The COBOL legacy program receives the transaction file name from the proxy program and proceeds with the batch update to an indexed sequential payroll master file. The update is driven by a transaction file which in this example deletes four records, adds the records back in, and then changes one of the records. The transaction file is shown below.

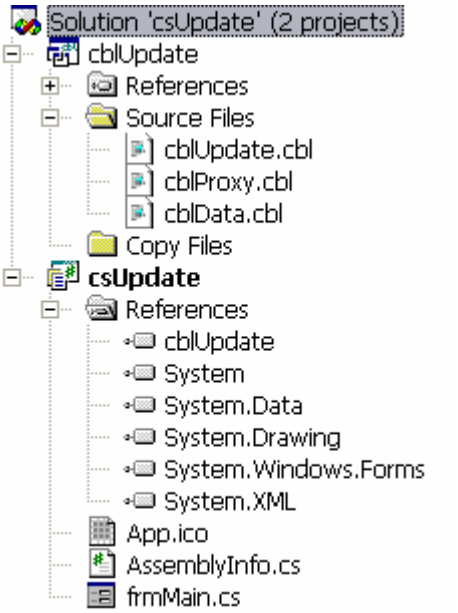


Following is a printout of the payroll master file after the update.



After the update, the legacy program returns the update statistics: records added, changed, deleted, and completion message back to the proxy program.

The execution of a legacy program within the .NET environment is within a .NET solution (csUpdate) that is comprised of two projects. The first project (csUpdate) is a C# project, and the second project (cblUpdate) contains three COBOL programs – the legacy program, the proxy program, and the data object class. The two projects must reference each other.



LEGACY PROGRAM

The functionality of the program is a batch update to an indexed-sequential file, in this case a payroll master file. The sequence of the update is driven by a text transaction file. Both the master and transaction files reside on the same drive and in the same folder. Since the name of the master file is static it is hard coded into the program. The transaction file is dynamic; therefore, the name of the file is supplied from the client program and routed through the proxy program to the legacy program. The update activity is written to a log file.

Some modifications must be made to the legacy program in order to communicate properly with the proxy program. These modifications have been underlined for easy reference. A linkage section is added to communicate between the proxy program and the legacy program. The Procedure Division statement is modified to reference the linkage section.

In the A100-Main paragraph an Exit Program statement replaces a Stop Run statement which would normally end a program. The Exit Program statement returns control to the proxy program after the execution of the program. In the B100-Initial paragraph the path and file name of the transaction file are placed into the filename field prior to opening the transaction file. In the B300-End paragraph the counts for the number of records added, changed, and deleted are placed into the Linkage Section fields, along with a message that the program is completed. Following is the code for the legacy program:

```
**********************************
*
* Legacy program for batch payroll update.
**********************************
*
* cblUpdate - This is a batch update to a
random access file.
*           The master file is a payroll file.
*           The update is driven by a transac-
tion file.
*           The activity is written to a log file.
**********************************
**

Identification Division.
Program-ID.          cblUpdate.

Environment Division.
Input-Output Section.
File-Control.
Select  100-Payroll Assign to "Payroll.dta"
       Organization is Indexed
       Access Mode is Random
       Record Key is 100-Emp-ID.

       Select  200-Transaction Assign to
        400-FileName
             Organization is line sequential.
       Select  300-Log Assign to "Log.dta"
             Organization is line sequential.
Data Division.
File Section.
Fd  100-Payroll.
01  100-Rec.
   05  100-Emp-ID          Pic 9(5).
   05  Filler              Pic X(47).

Fd  200-Transaction.
01  200-Rec                Pic X(53).

Fd  300-Log.
```

```
01  300-Rec                 Pic X(60).

Working-Storage Section.
01  400-Work-Fields.
   05  400-Action           Pic X     Value "
".
      88  No-More                    Value
"x".
   05  400-There-Flag       Pic X     Value "
".
      88  Record-There               Value
"Y".
      88  Record-Not-There           Value
"N".
   05  400-FileName         Pic X(80) Value "
".
   05  400-Record-Counts.
      10  400-Add-Count     Pic 9(6)  Value 0.
      10  400-Chg-Count     Pic 9(6)  Value 0.
      10  400-Del-Count     Pic 9(6)  Value 0.



01  500-Work-Rec.
   05  500-Emp-ID           Pic 9(5).
   05  500-TC               Pic X.
   05  500-First-Name       Pic X(12).
   05  500-Last-Name        Pic X(15).
   05  500-Department       Pic X(12).
   05  500-Hours            Pic 99V99.
   05  500-Hours-X          Redefines 500-
                            Hours
                            Pic X(4).
   05  500-Rate             Pic 99V99.
    05  500-Rate-X          Redefines 500-
Rate
                            Pic X(4).
 01  600-Work-Rec.
    05  600-Emp-ID          Pic 9(5).
    05  600-First-Name      Pic X(12).
    05  600-Last-Name       Pic X(15).
    05  600-Department      Pic X(12).
    05  600-Hours           Pic 99V99.
    05  600-Rate            Pic 99V99.

Linkage Section.
01 lnk-FileName             Pic X(80).
01 lnk-addCount             Pic 9(6).
01 lnk-chgCount             Pic 9(6).
01 lnk-delCount             Pic 9(6).
01 lnk-txtMessage           Pic X(50).


Procedure Division using lnk-FileName
        lnk-addCount
        lnk-chgCount
        lnk-delCount
        lnk-txtMessage.

A100-Main.
    Perform B100-Initial.
    Perform B200-Process Until No-More.
    Perform B300-End.
    Exit    Program.

B100-Initial.
    Move    lnk-FileName to 400-FileName.
    Open    I-O    100-Payroll.
    Open    Input  200-Transaction.
    Open    Output 300-Log.
    Perform C200-Get-Transaction-Record.

B200-Process.
    Perform C100-Get-Payroll-Record.
    Perform C300-Update.
    Perform C200-Get-Transaction-Record.

B300-End.
    Close   100-Payroll.
    Close   200-Transaction.
    Move    "Program Ended!" to 300-Rec.
    Write   300-Rec.
    Close   300-Log.
    Move    400-Add-Count to lnk-addCount.
    Move    400-Chg-Count to lnk-chgCount.
    Move    400-Del-Count to lnk-delCount.
    Move    "Update completed successfully!"
to
            lnk-txtMessage.

 C100-Get-Payroll-Record.
    Move    500-Emp-ID to 100-Emp-ID.
    Move    "Y" to 400-There-Flag.
    Read    100-Payroll Into 600-Work-Rec
             Invalid Key Move "N" to
             400-There-Flag
    End-Read.

C200-Get-Transaction-Record.
    Read    200-Transaction Into 500-Work-
Rec
             At End Move "x" to 400-Action
    End-Read.

C300-Update.
    If      500-TC is equal to "a"
    If      400-There-Flag is equal to "Y"
           Move   space to 300-Rec
           String 500-Emp-ID " Already on
file"
             into 300-Rec
             Write  300-Rec
         Else
            Perform D100-Add
```

```
            Exit Paragraph
        End-IF.
        If      500-TC is equal to "c"
          If    400-There-Flag is equal to "N"
                Move   space to 300-Rec
                String 500-Emp-ID " Not on file"
                  into 300-Rec
                Write  300-Rec
           Else
                Perform D200-Change
                Exit Paragraph
        End-IF.
        If      500-TC is equal to "d"
          If    400-There-Flag is equal to "N"
                Move   space to 300-Rec
                String 500-Emp-ID " Not on file"
                  into 300-Rec
                Write  300-Rec
           Else
                Perform D300-Delete
                Exit Paragraph
        End-IF.

D100-Add.
        Move    500-Emp-ID     to
                  600-Emp-ID 100-Emp-ID.
        Move    500-First-Name to
                  600-First-Name.
        Move    500-Last-Name  to
                  600-Last-Name.

        Move    500-Department to
                  600-Department.
        Move    500-Hours      to 600-Hours.
        Move    500-Rate       to 600-Rate.
        Move    " " to 400-Action.
        Write   100-Rec from 600-Work-Rec
              Invalid Key
              Move       "e" to 400-Action
              Move       "Error on Add" to 300-
Rec
              Write      300-Rec
        End-Write.
        If      400-Action not = "e"
              Move   space to 300-Rec
              String 500-Emp-ID " Record
Added"
                    into 300-Rec
              Write  300-Rec
              Add    1 to 400-Add-Count
        End-If.

D200-Change.
        If      500-First-Name
                  is not equal to space
              Move 500-First-Name to
              600-First-Name
        End-If.
        If      500-Last-Name is not equal to
space
              Move 500-Last-Name to
              600-Last-Name
        End-If.
        If      500-Department
              is not equal to space
              Move 500-Department to
              600-Department
        End-If.
        If      500-Hours-X is not equal to space
              Move 500-Hours to 600-Hours
        End-If.
        If      500-Rate-X is not equal to space
              Move 500-Rate to 600-Rate
        End-If.
        Move    " " to 400-Action.
        Rewrite 100-Rec from 600-Work-Rec
              Invalid Key
              Move    "e" to 400-Action
              Move   space to 300-Rec
              String 500-Emp-ID
              " Error on Change"
                    into 300-Rec
              Write  300-Rec
        End-Rewrite.
        If      400-Action is not equal to "e"
               Move   space to 300-Rec
              String 500-Emp-ID
              " Record Changed"
                    into 300-Rec
              Write  300-Rec
              Add    1 to 400-Chg-Count
        End-If.

D300-Delete.
        Move    500-Emp-ID to 100-Emp-ID.
        Move    " " to 400-Action.
        Delete  100-Payroll
              Invalid Key
              Move    "e" to 400-Action
              String 500-Emp-ID
              " Error on Delete"
                    into 300-Rec
              Write  300-Rec
        End-Delete.
        If      400-Action is not equal to "e"
              String 500-Emp-ID
              " Record Deleted"
                    into 300-Rec
              Write  300-Rec
              Add    1 to 400-Del-Count
         End-If.
```

PROXY (WRAPPER) PROGRAM

The Proxy or Wrapper program is the link between the Client (C#) program and the legacy (COBOL) program. The Proxy program also interacts with the Data Object class. The Proxy program creates an instance of the Data Object in the Repository which is referenced in the Linkage Section. The Repository provides the means of relating internal names with external references. The Proxy program receives the name of the transaction file from the Client program and returns the Data Object back to the Client program. A Local-Storage section, which is visible only within the method, is used to contain the name of the transaction file, the update counts, and the completion message which are returned from the legacy program. The Local-Storage functions as the common data area with the legacy program. The Transaction file name is placed from the Linkage Section into the Local-Storage section, and then the legacy program is called. At the conclusion of the execution of the legacy program an instance of the Data Object is created, and the values are sent from the Local-Storage section into the Data Object. At the conclusion of the Proxy program control is returned back to the Client program.

```
**********************************
*
* Proxy class for batch payroll update.
**********************************
*
* cblProxy - This program provides the
* functionality to send a file name to the
* batch update program (cblUpdate)  and
* return counts for the number of records
* added, changed, and deleted, and a
* message after the program is run.
**********************************
*
 CLASS-ID. cblProxy As "cblProxy".
 REPOSITORY.
    CLASS StringClass As "System.String"
    CLASS dataObject As "dataObject"  .
 STATIC.
 DATA DIVISION.
 PROCEDURE DIVISION.
 METHOD-ID. CallUpdateProgram As
         "CallUpdateProgram".
 Local-Storage Section.
 01  lo-DataIn.
    05  lo-FileName          Pic X(80).
 01  lo-DataOut.
    05  lo-Message           Pic X(50).
    05  lo-Added             Pic 9(6).
    05  lo-Changed           Pic 9(6).
    05  lo-Deleted           Pic 9(6).

 LINKAGE SECTION.
 01 lnk-FileName Object Reference String-
Class.
 01 lnk-dataObj  Object Reference dataOb-
ject.

 PROCEDURE DIVISION Using by value
                  lnk-FileName
                  Returning     lnk-dataObj.
      SET lo-FileName to lnk-FileName.
      CALL "cblUpdate" USING lo-FileName
                        lo-Added
                        lo-Changed
                        lo-Deleted
                        lo-Message  .

 Invoke dataObject "New" Returning
          lnk-DataObj.
 Set    lnk-dataObj::"ReturnMessage" to
          lo-Message.
 Set    lnk-dataObj::"ReturnAdded"   to
          lo-Added.
 Set    lnk-dataObj::"ReturnChanged" to
          lo-Changed.
 Set    lnk-dataObj::"ReturnDeleted" to
          lo-Deleted.

 END METHOD CallUpdateProgram.
 END STATIC.
 END CLASS cblProxy.
```

DATA OBJECT CLASS

The Data Object class serves only one purpose, to store data that is transferred from the Proxy program back to the Client program. The data consists of a message and three counts for the number of records added, changed, or deleted in the payroll master file. The message indicates that the update completed successfully. The property identifies the class data member that is referenced in the Client program. A property is a name that is used to qualify an object reference so a value can be passed into or out of the object. This allows the transference of data that is defined in a COBOL program as Picture X or Picture 9 easily into a C# program.

```
**********************************
* Data Object Class.
```

```
**********************************
* cblData - This class defines the data object
* which is used to pass data from the wrap-
per * to the client.
**********************************
 Identification Division.
 Class-id. dataObject As "dataObject".
 Environment Division.
 Configuration Section.
 Repository.
    Class DecimalClass as "System.Decimal"
    Class StringClass as "System.String"  .
  Object.
 Data Division.
 Working-Storage Section.
 01  ReturnMessage
    Object Reference StringClass
    Property as "ReturnMessage".
 01  ReturnAdded
    Object Reference DecimalClass
    Property as "ReturnAdded".
 01  ReturnChanged
    Object Reference DecimalClass
    Property as "ReturnChanged".
 01  ReturnDeleted
    Object Reference DecimalClass
    Property as "ReturnDeleted".
 End Object.
 END CLASS dataObject.
```

CLIENT PROGRAM

The Client program is written in C#, although it could just as easily have been written in Visual Basic. A class string data member is defined to hold the path and filename of the transaction data file which is populated using a common dialog box.

In the update method an instance of the Data Object is created. Then the COBOL Proxy program is called by sending the transaction file name and returning the update counts and message into an instance of the Data Object. The Class-ID is named cblProxy and the Method-ID is named CallUpdateProgram. Finally, the message and update counts are assigned to the appropriate textboxes from the data members of the Data Object. The counts, being numeric, must be converted to string. Only the relevant code of the C# program is shown below.

```
/*
 **********************************
 A C# program that provides a Graphical
User Interface (GUI) to the legacy program.
This
program uses a common dialog box to get
the path and file name of the transaction
file.

The path and file name are passed to the
Proxy class.

The Proxy class returns counts for records
added, changed, and deleted, and a mes-
sage.
 **********************************
 */
public class csMain : Sys-
tem.Windows.Forms.Form
{
        private string strFileName = "";
        .  .  .  .


private void btnUpdate_Click(object sender,
System.EventArgs e)
{
  dataObject dtaObj = new dataObject();
  dtaObj =
cblProxy.CallUpdateProgram(strFileName);
  txtMessage.Text = dtaObj.ReturnMessage;
  txtAdded.Text = Con-
vert.ToString(dtaObj.ReturnAdded);
  txtChanged.Text = Con-
vert.ToString(dtaObj.ReturnChanged);
  txtDeleted.Text = Con-
vert.ToString(dtaObj.ReturnDeleted);
}
```

SUMMARY AND CONCLUSIONS

Since COBOL programs will be part of business information systems for the foreseeable future, making COBOL part of contemporary development environments is a natural consequence. The focus of this study has been the incorporation of COBOL into Microsoft's .NET Framework. Within that framework, old legacy COBOL programs can be integrated with contemporary languages such as Visual Basic or C#. The process is accomplished by creating an application in a .NET solution that consists of two projects. The first project contains the .NET user interface, in this case written in C#. The second project contains two required COBOL programs, and one optional program. The required

programs are the legacy program and a proxy, or wrapper, class that is the link between the legacy program and the user interface program. A third COBOL program is an object class that is used to transfer data between the C# program and the COBOL proxy class. There are other means of transferring data but this is the most efficient, so it is not optional in this case.

Only slight modifications to the legacy COBOL program are required to enable communication between the legacy program and the proxy class. The proxy class is the heart of the interface. This COBOL object class calls the COBOL legacy program, and in turn is called by the C# program. The transference of data between the legacy program and the proxy class is via normal COBOL processing with a Linkage Section. The transference of data between the proxy class and the C# program is accomplished with a data object class. An instance of the data object is created in both the proxy class and the client (C#) program. Data is placed into the data object in the proxy class and then retrieved by the client program. The result is the execution of a legacy COBOL program 'by proxy.'

The most appropriate location to place this type of project into the information systems curriculum would be an advanced programming course. At this level the students would have a good background in the .NET environment and a working understanding of updating a master file. This would ease the process of interfacing .NET with a legacy COBOL program. While understanding the function of the legacy program to make modifications is necessary, a full understanding of COBOL would not be required.

REFERENCES

Malek, Rick. "Why Object Orientation for COBOL?" (2002), Online: http://www.c-sharpcorner.com/Code/2002/July/Art02-OOIntro.asp

Malek, Rick. "Calling Procedural COBOL from VB.NET," (2002) Online: http://www.c-sharpcorner.com/Code/2002/Aug/vb2cobol.asp

Malek, Rick. "Welcome to COBOL.NET Corner," (2002) Online: http://www.c-sharpcorner.com/Code/2002/June/WelcomeCobolNet.asp

Micro Focus. (2003). Net Express With .NET. Beaverton, OR.: Micro Focus International Limited.

Price, Wilson T. (1997). Elements of Object-Oriented COBOL. Orinda, CA.: Object-Z Publishing.

Price, Wilson T. & Rippin, Wayne. (2004). COBOL and .NET. Draper, UT.: Object-Z Publishing.

Sharp, John & Jagger, Jon. (2002). VISUAL C# .NET. Redmond, WA.: Microsoft Press.

Surapaneni, Narayana Rao. "COBOL for Microsoft .NET, "(2001) Online: http://www.c-sharpcorner.com/CobolNet/Cobol4MSNETNRS.asp