The COBOL jigsaw puzzle: Fitting object-oriented and legacy applications together

by E. S. Flint

Object wrappers have been presented as a way to allow legacy applications and object-oriented applications to work together. However, object wrappers do not always solve the interoperability problem for COBOL legacy applications. This paper examines the use of object wrappers and introduces two other types of wrappers, the procedural wrapper and the combination wrapper, for practical use with COBOL legacy applications. The main concerns of a developer of an object-oriented application that uses the services of or provides services to a legacy application are addressed. Examples of "real-world" COBOL legacy applications are cited and samples of all three types of wrapper code are provided.

A legacy COBOL application can be characterized as procedure-oriented, with perhaps several legacy programs working together. It can be a puzzle to get several legacy programs to run as one unit, and getting a legacy program and an object-oriented program to run together can be compared to a jigsaw puzzle. Like the jigsaw puzzle, the final picture can be formed only if the correct connections are made between the pieces. In this paper, three types of "wrapper" programs are presented as techniques for making the connections between COBOL legacy programs and object-oriented programs.

Early work proved the applicability of wrappers for using legacy code as an external object implementation.[1] In this paper, a wrapper is considered to be a "shell" program that hides implementation details of the legacy program from the object-oriented program and provides an interface between the two programs. This paper describes three types of wrappers: object-oriented, procedural, and combination. With one of these wrappers, a COBOL legacy application is able to work with an object-oriented application. Further, using these wrappers requires a *minimum number of changes to a legacy application.*

The first section explains why COBOL legacy programs and object-oriented programs do not easily form interoperable components. Next, a section is devoted to each of the three wrappers to explain what the wrapper is, discuss when it should be used, and provide an example of its use. Sample code is also provided to illustrate the use of each of the wrappers. Finally, the conclusion summarizes the benefits of using wrappers.

Background

Today most commercial and government enterprises are becoming increasingly aware of object technology and the benefits resulting from its effective deployment. Benefits include:

- Increased programmer productivity
- Faster development of applications
- Reuse of models, designs, and code
- Preserving existing applications

The last item has generated a great deal of interest and concern since the introduction of IBM COBOL for MVS & VM, IBM VisualAge* for COBOL for OS/2*, and IBM COBOL Set for AIX*, which are the new object-oriented COBOL compilers from IBM.

With an estimated 200 billion lines of code in existence, COBOL is by far the most widely used programming language.[2] Enterprises are dependent on existing, large-scale COBOL applications—legacy COBOL applications. Most enterprises with large investments in COBOL code are anxious to move forward with state-of-the-art object technology, yet dread the cost of reprogramming legacy applications that have been used successfully for many years.

The conflict between legacy and object-oriented applications is in their method of operation.

It is difficult to ignore a technology that offers productivity and reuse benefits. However, it is equally difficult to consider rewriting legacy applications that currently work. The conflict between legacy and object-oriented applications is in their method of operation. The legacy application is a linear block of code with a sequence of PERFORM and CALL statements. The object-oriented application is a fluid, data-centered collection of classes. It creates object instances and directs messages to their methods, requesting service. Can two types of applications with such different modes of operation ever work together? If so, what are the fundamental issues in designing and building the interoperable components?

This paper addresses these questions through wrappers. A wrapper is code that provides an interface for one program to access the functionality of another program. For the purpose of this paper, one program is a COBOL legacy program and the other is an object-oriented program. The specific responsibilities of a wrapper are to provide translation between the invocation format of an object-oriented application and the call format of a legacy application, and to provide data in the appropriate form for the object-oriented application and the legacy application.

Given this general definition of a wrapper, the three wrappers discussed in this paper are defined as follows:

An object wrapper is an object that acts as a "front end" to a legacy application, transforming its functional interface to an object interface.

A procedural wrapper is a program that acts as a "back end" to a legacy application, transforming its functional interface into an object interface.

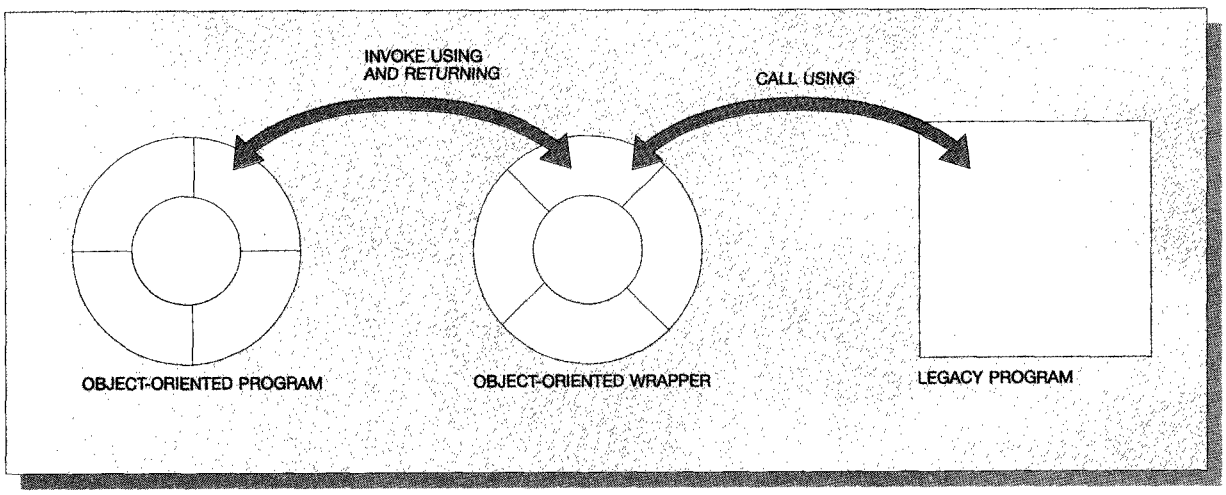
A combination wrapper is a program that instantiates one or more objects, all of which act as back ends to a legacy application, transforming its functional interface into an object interface.

Other papers have described how object wrappers allow an object-oriented application to use the services of a legacy application. In this case, the object-oriented application is the client, or main program.[3] Unfortunately, many COBOL applications require the legacy program to be the client and to use services provided by the object-oriented program. For example, a new function, which traditionally is added as a new subroutine, can be added as an object. But now the legacy program must create an instance of the object and direct messages to the methods, resulting in many changes to the legacy program. Since the object wrapper is itself an object, nothing is gained by attempting to use it in this situation. The legacy program would have to create an instance of the object wrapper and send messages to it. Thus, we need to consider two other types of wrappers. With the procedural wrapper and the combination wrapper, minimal changes are necessary for the COBOL legacy program to remain the client while enhancements are added as object-oriented programs.

Object-oriented wrapper

An object-oriented wrapper, or object wrapper as it is often called, is an object that encapsulates a COBOL legacy application, transforming its functional interface to an object interface. Methods for the wrapper are written and packaged as an object-oriented COBOL class. Other programs can reuse existing legacy programs by instantiating objects from the object wrapper class or its descendants. The functional interface and data structures of the legacy ap-

Figure 1 Object-oriented wrapper



plication are hidden from other programs, and it looks and acts like another object in the system.

The object-oriented wrapper is useful when a legacy program becomes the server in a client/server application. Consider a legacy program that has an archaic command-line user interface. To use the program in a client/server environment, a graphical user interface (GUI) is added, written as an object-oriented program. This GUI program acts as the client program and the legacy program becomes the server program, responding only to requests. Unfortunately the objected-oriented GUI program sends messages and passes large blocks of data, neither of which the legacy program is equipped to handle.

The object-oriented wrapper solves this communication problem by serving as the "translator" for both the messages and the data structures. The object-oriented wrapper must establish data structures for:

- Data passed from the object-oriented program
- Data passed to the legacy program
- Data returned from the legacy program
- Data returned to the object-oriented program

An object-oriented wrapper organizes the messages and data from the object-oriented program into a form the legacy program can handle, as illustrated in Figure 1.

Writing an object-oriented wrapper. There are two steps in writing an object-oriented wrapper.

First, set up the necessary data structures. Determine all the data that must be passed between the object-oriented and the legacy programs. Divide the data into two categories: the data that are always needed, regardless of the task, and the data that are specific to a particular task.

The data that are always needed are defined as instance data in the wrapper. The data that are specific to a particular task are coded in the LINKAGE SECTION of the appropriate method in the wrapper program.

Next, perform a task analysis of the legacy program to determine all of its functions. Usually, each task corresponds to a method in the object-oriented wrapper. The code in a typical method:

1. Contains LOCAL-STORAGE SECTION data items necessary for the method to complete its task
2. Contains LINKAGE SECTION data items necessary for passing parameters and return values
3. Parses or translates data items passed from the object-oriented program
4. CALLs the legacy program. (This can be a CALL to the main program, a subprogram, or an entry point.)
5. Parses or translates data items returned from the legacy program

6. Returns to the object-oriented program

Example. Consider the command-line legacy program and GUI object-oriented program discussed earlier. Let us assume that the legacy program does database searches and has a group of subprograms, each of which gets data for a particular type of search. One such subprogram might be called *NameSearch* and includes the following series of DISPLAY and ACCEPT statements:

```
display 'What is the name?'.
accept name.
display 'Search for department (D) or phone (P)?'.
accept criteria.
call 'SearchIt' using name criteria result.
display 'The result is ' result.
```

On the GUI side, a user enters data in entry fields and presses the push button for the type of search. The object-oriented GUI program collects all the data from the screen and sends a *Search_Name* message.[4] The object-oriented wrapper intercepts the message, prepares the data, and CALLs the search subprogram. When the legacy program finishes the search, control is returned to the wrapper, which prepares the data and returns control to the GUI program. The code for the object-oriented wrapper is shown in Figure 2.

What modifications need to be made to the legacy program? In Figure 2, the wrapper program makes the CALL directly to the subprogram, *SearchIt*, bypassing NameSearch. Alternatively, NameSearch could be CALLed if the DISPLAY and ACCEPT statements were commented out or removed. If a legacy program is not organized into logical, structured subprograms, it may need to be restructured or have entry points added.

Procedural wrapper

A procedural wrapper is a program that reconciles a COBOL legacy application's functional interface to an object interface. Modules are written and packaged as entry points in a COBOL program, which is the procedural wrapper. Existing legacy programs can reuse object-oriented programs by calling the appropriate entry point in the procedural wrapper. The invocation interface and data structures of the object-oriented application are hidden from the legacy application, which sees a functional interface that looks like another subroutine in the system.

In the COBOL environment, legacy programs must frequently continue to be the main, or client, programs. Often a legacy program needs either an existing function changed or a new function added, but this function is a subprogram, or server, to the legacy program. Often the function can be modeled as a class, resulting in an object-oriented program. Now the legacy program is trying to CALL the function modeled by the class, but the object instantiated from the class responds only to messages.

This problem can be solved by a procedural wrapper if the following conditions are true:

- The number of methods in the class is predictable. The object-oriented program is stable; methods are added or deleted either rarely or not at all. The hierarchy of subclasses is stable and the probability of adding new subclasses is very low. This results in a stable procedural wrapper that can be used by multiple legacy applications.
- A small number of data items are shared between the legacy program and the object-oriented program. The size of the LINKAGE SECTION in the procedural wrapper depends on how many different types of data items must be passed as parameters. If there are many different types of data items, a combination wrapper should be considered (see Figure 3).

A procedural wrapper allows the legacy program to continue as the main program by answering CALLs from the legacy program and sending INVOKE messages to the object-oriented program as illustrated in Figure 4. The parameters passed to the procedural wrapper are (1) a data item or data structure (required) and (2) a status flag (optional).

Writing a procedural wrapper. To write a procedural wrapper:

1. Identify all the data items that must be passed between the legacy and object-oriented programs. The data are coded in the LINKAGE SECTION of the wrapper program.
2. Match each method in the object-oriented program to an entry point in the wrapper program. The code after the ENTRY statement:
   a. Creates an instance of the object
   b. Parses or translates data items passed from the legacy program
   c. INVOKEs the appropriate method
   d. Parses or translates data items returned from the object-oriented program
   e. Frees the instance of the object
   f. Returns to the legacy program

Figure 2 Object-oriented wrapper code

```
Identification Division.
Class-Id.  Wrapper1 inherits SOMObject.
Environment Division.
Configuration Section.
Repository.
    Class Wrapper1 is 'Wrapper1'
    Class SOMObject is 'SOMObject'.
Data Division.
Working-Storage Section.
*  Data structures for the class
Procedure Division.

Identification Division.
Method-Id.  'Search_Name'.
*
*  Data structures for this specific task
Data Division.
*
*  Data items parsed by wrapper & passed to/from legacy program
Local-Storage Section.
01  name pic x(20).
01  result pic x(25).
01  criteria pic x(1).
*
*  Data structures passed to/from object-oriented program
Linkage Section.
01  input-data pic x(80).
01  output-data pic x(80).
Procedure Division using input-data returning output-data.
*
*  Parse data structure received from object-oriented program
    move input-data(1:20) to name.
    move input-data(21:1) to criteria.
*
*  Call the legacy program, in this case a subprogram
    call 'SearchIt' using name criteria result.
*
*  Translate data items returned from legacy program
    move spaces to output-data.
    move result to output-data(1:25).
*
*  Return to the object-oriented program
    exit method.
End Method 'Search_Name'.

*  Other methods to handle messages from the GUI Interface program.
End Class Wrapper1.
```

3. Add other data items and code to complete the wrapper. Include:
   a. A class definition in the REPOSITORY PARAGRAPH
   b. One or more OBJECT REFERENCE data items

Example. A major concern today is the "year 2000" problem. Consider a legacy program with the following lines of code:

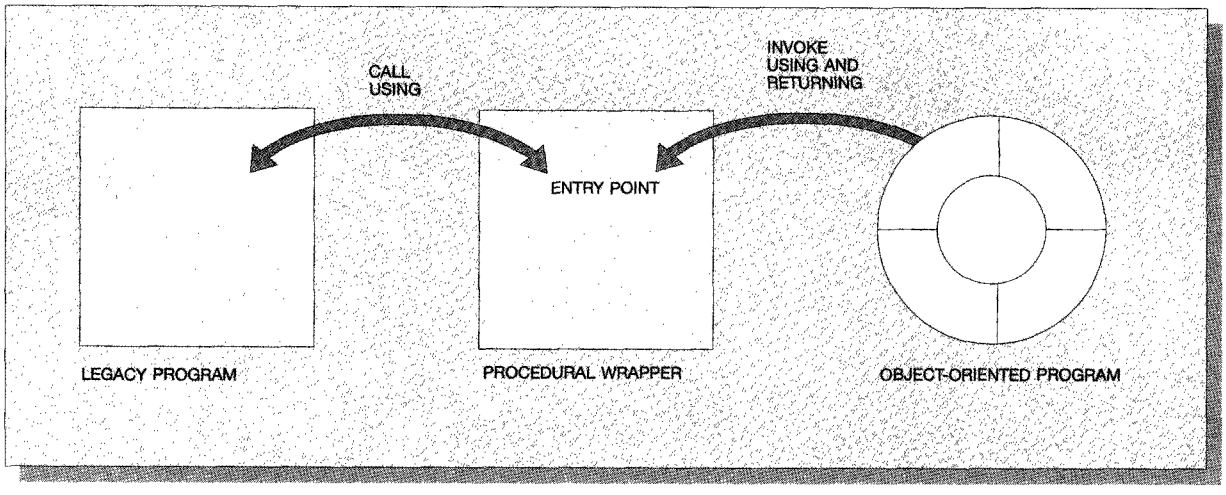
```
accept gregorian from date.
move corresponding gregorian to edited-gregorian.
```

The date obtained here has a two-digit year. However, we now need a date with the four-digit year.

Assume that we have an object-oriented program that determines a Gregorian date, a Julian date, or a Lilian date, all of which have a four-digit year. The class name is *Year2000Date* and its methods have the following signatures:

GregorianDate	returns PIC 9(8)
JulianDate	returns PIC 9(7)
LilianDate	returns PIC 9(7)

Figure 3 Procedural wrapper



A procedural wrapper will allow a legacy program to use the object-oriented program. The wrapper will have an ENTRY statement that corresponds with each method in the class and a data item that corresponds with each data item returned from the object-oriented program. The code for the procedural wrapper is shown in Figure 4.

What changes are required to the legacy program? In the example legacy program, the ACCEPT statement will be replaced by the following statement:

```
call 'AcptDate' using gregorian.
```

Also, all data declarations for the year must be changed from PIC 99 to PIC 9999. If the date data declarations are in a COPYLIB, this change is relatively easy.

Combination wrapper

A combination wrapper is a program that instantiates one or more objects, all of which reconcile a COBOL legacy application's functional interface to an object interface. The combination wrapper is made up of two parts: the procedural portion and the object-oriented portion. Modules are written and packaged as entry points in a COBOL program, the procedural portion. Methods are written and packaged as object-oriented COBOL classes, the object-oriented portion. Existing legacy programs can reuse object-oriented programs by calling the appropriate entry point in the procedural portion, which creates the appropriate objects in the object-oriented portion. The invocation interface and data structures of the object-oriented application are hidden from the legacy application. As with the procedural wrapper, the object-oriented applications look like another subroutine to the legacy application.

The combination wrapper is more effective than the procedural wrapper for larger applications. The object portion is essentially a hierarchy of object wrappers. The highest level is a basic object wrapper that provides only the most general methods. At lower levels are specific object wrappers, each abstracting properties of legacy-to-object-oriented application interaction. If a COBOL legacy program needs to use services provided by an object-oriented application and a stable combination wrapper exists, then one of the following applies:

- One of the existing objects in the object-oriented portion can be reused.
- A subclass can be created from one of the existing objects in the object-oriented portion.

The "year 2000" problem is easily solved with a procedural wrapper because the object-oriented program has a fixed number of methods and a small number of shared data items. If the legacy program requires that many different data structures be shared with the object-oriented program, then a combination wrapper is easier to design and maintain.

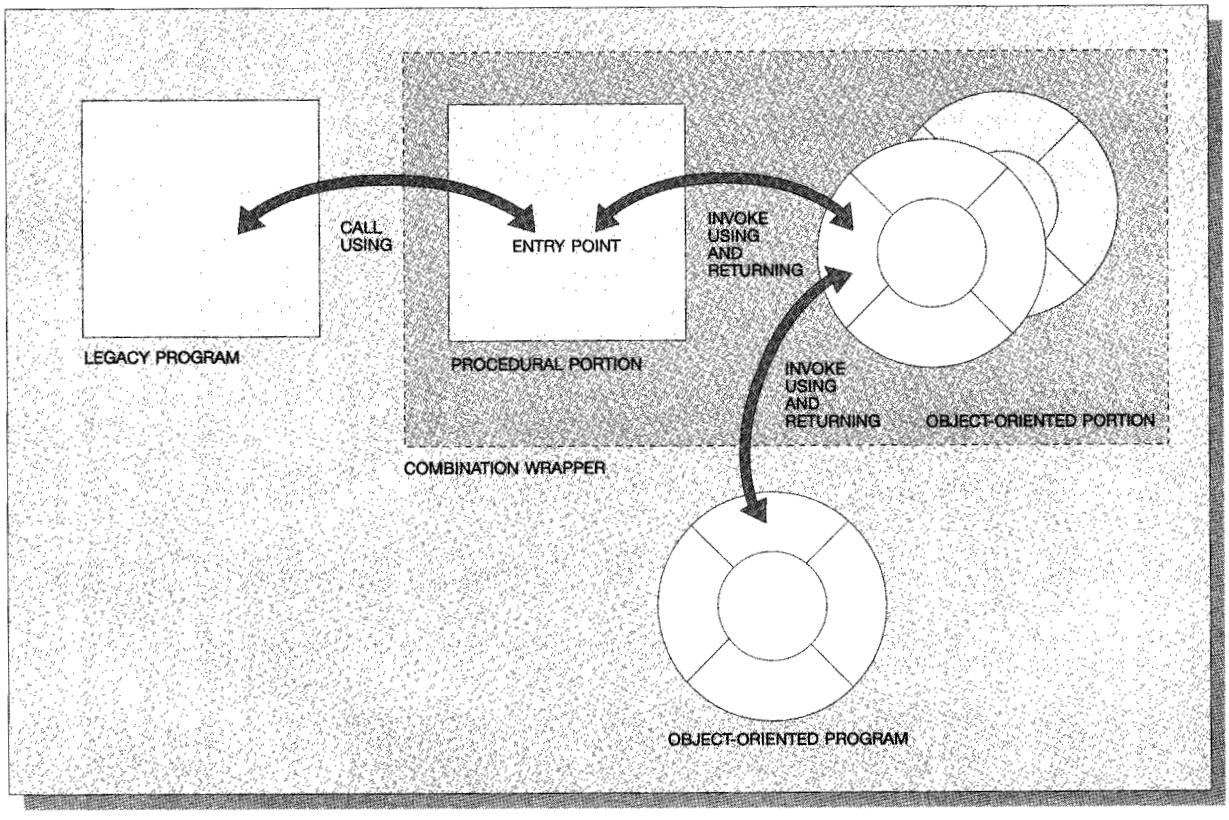
Figure 4 Procedural wrapper code

```
Identification Division.
Program-Id. 'Wrapper2'.
Environment Division.
Configuration Section.
Repository.
    Class Year2000Date is 'Year2000Date'.
Data Division.
Working-Storage Section.
*
*  Data items needed by procedural wrapper
01  anObj object reference value null.
Linkage Section.
*
*  Data items passed between legacy and object-oriented programs
01  gregorian pic 9(8).
01  julian    pic 9(7).
01  lilian    pic 9(7).
Procedure Division.
*
*  Entry point for Gregorian date
*
    entry 'AcptDate' using gregorian.
*
*  Create instance of object
    invoke Year2000Date 'somNew' returning anObj.
*
*  Invoke the method
    invoke anObj 'GregorianDate' returning gregorian.
*
*  Free instance of object
    invoke anObj 'somFree'.
*
*  Return to legacy program
    exit program.
*
*
*  Entry point for Julian date
*
    entry 'AcptDay' using julian.
    invoke Year2000Date 'somNew' returning anObj.
    invoke anObj 'JulianDate' returning julian.
    invoke anObj 'somFree'.
    exit program.
*
*
*  Entry point for Lilian date
*
    entry 'AcptInt' using lilian.
    invoke Year2000Date 'somNew'  returning anObj.
    invoke anObj 'LilianDate' returning lilian.
    invoke anObj 'somFree'.
    exit program.

End program 'Wrapper2'.
```

The combination wrapper combines the procedural wrapper and the object-oriented wrapper as illustrated in Figure 5. The procedural portion of the combination wrapper takes CALLs from the legacy program and creates the appropriate object instance from the object-oriented portion of the combination wrapper. The parameters passed to the procedural portion are the subclass name (required), a data structure (optional, depending on action), and a status flag (optional).

Figure 5 Combination wrapper



The object-oriented portion of the combination wrapper is an inheritance hierarchy in which the parent is an abstract class. Each child is a concrete class that corresponds to a data structure used in a legacy program. Methods in a child correspond to the various tasks performed on the data structure. The parameters passed to the object-oriented portion are a POINTER to the data structure (required) and a status flag (optional).

Writing a combination wrapper. It is usually easier to start with the object-oriented portion of the combination wrapper. To write the subclasses, or child programs:

1. Identify all the different data structures that exist in the legacy program. Each data structure defines a subclass. To make maintenance easier, give the subclass a name similar to the name of the program and the function of the data structure. For example, if *PROGRAM1* has an input data structure, name the subclass that manipulates this data structure *ProgOneInput*. These data structures must be coded in the subclass LINKAGE SECTION.
2. Identify the actions performed on or with each of the data structures from the legacy program. Each action on a data structure defines a method in the subclass. The code in a typical method in the object-oriented portion of the wrapper does the following:
   a. If necessary (OPEN, for example), creates an instance of the object
   b. Sets ADDRESS OF the data structure in the LINKAGE SECTION to the POINTER passed from the procedural portion of the combination wrapper
   c. Establishes the data for parameters passed to the object-oriented program
   d. INVOKES the appropriate method
   e. Parses or translates data items returned from the object-oriented program

Figure 6 Combination wrapper code—object-oriented portion (part 1 of 5)

```
 Identification Division.
*  Abstract parent class
 Class-Id.  Wrapper3ObjectOriented inherits SOMObject.
 Environment Division.
 Configuration Section.
 Repository.
     Class Wrapper3ObjectOriented is 'Wrapper3ObjectOriented'
     Class SOMObject is 'SOMObject'.
 Procedure Division.
*
*  General methods
*
 End Class Wrapper3ObjectOriented.
```

f. If necessary (CLOSE, for example), frees the instance of the object
g. Returns to the procedural portion of the wrapper

Usually, writing the procedural portion of the combination wrapper is similar to writing a procedural wrapper. There are some differences:

- A large data item, for example PIC X(1000), must be defined in the wrapper's LINKAGE SECTION to hold the largest possible data structure that can be passed from the legacy program. The use of a large, generic data item shields the wrapper from knowing the exact data structures it is passing. A POINTER is used to pass the generic data item between the procedural and the object-oriented portions of the combination wrapper.
- A large table must be defined in the wrapper's LOCAL-STORAGE SECTION to hold subclass names and their OBJECT REFERENCEs. Each legacy program that uses the wrapper has an entry point, called a *program entry point*, in the wrapper. Each unique action in the legacy program has an entry point, called an *action entry point*, in the wrapper.

The code after the ENTRY statement in the procedural portion does the following:

- A program entry point (1) creates an instance of the appropriate subclass, (2) puts the subclass name and OBJECT REFERENCE in the table, and (3) returns to the legacy program.
- An action entry point (1) sets a pointer to the address of the data structure passed from the legacy program, (2) finds the subclass name in the table, (3) INVOKEs the appropriate method in the subclass using the OBJECT REFERENCE from the table, (4) if necessary (CLOSE, for example), frees the instance of the subclass and cleans up the table entry, and (5) returns to the legacy program.

Subclass definitions in the REPOSITORY PARAGRAPH complete the procedural portion code.

Example. Suppose an old file system is to be replaced by a CD-ROM file system with an object-oriented interface. And a large group of legacy programs that used the old file system now must be migrated to the CD-ROM system. It is likely that every legacy program in the group has one or more different record structures. One program might have only *name* and *address* fields in its record. Another program might have *name*, *age*, and *salary* fields. Since these two programs have different data structures, the result is two different subclasses in the object-oriented portion of the combination wrapper.

Let us assume that the first program, *ProgOne*, sequentially READs the entire file. Thus, the methods in the first subclass, *ProgOneInput*, are *open*, *read*, and *close*. Also, let us assume that the second program, *ProgTwo*, sequentially WRITEs records to a file. So the methods in the second subclass, *ProgTwoOutput*, are *open*, *write*, and *close*. The code for the object-oriented portion of the combination wrapper is shown in Figure 6.

The procedural portion of the combination wrapper has an ENTRY statement to correspond to each legacy program and each unique action in the group

Figure 6 Combination wrapper code—object-oriented portion (part 2 of 5)

```
Identification Division.
*  Concrete child class for Program One's input
Class-Id. ProgOneInput inherits Wrapper3ObjectOriented.
Environment Division.
Configuration Section.
Repository.
    Class CDROM is 'CDROM'
    Class ProgOneInput is 'ProgOneInput'
    Class Wrapper3ObjectOriented is'Wrapper3ObjectOriented'.
Data Division.
Working-Storage Section.
01  fileObj object reference.
Procedure Division.

Identification Division.
*  Read action
Method-Id.  'Read'.
Data Division.
Local-Storage Section.
01  file-name pic x(20) value spaces.
Linkage Section.
*
*  Input data structure
01  file-record pic x (100).
*
*  Data items passed between legacy and object-oriented programs
01  file-ptr pointer.
01  file-flag pic x.
    88 eof value '1'.
Procedure Division using file-ptr returning file-flag.
*
*  Set address for the input data structure
    set address of file-record to file-ptr.
*
*  Establish parameter for object-oriented program
     move z'FLIGHT-FILE' to file-name.
*
*  Invoke method
    invoke fileObj 'ReadSeqFile'
         using by content file-name
               by reference file-record
         returning file-flag.
*
*  Return to procedural portion
    exit method.
End Method 'Read'.
```

of legacy programs. The unique actions in these two legacy programs are OPEN, READ, WRITE, and CLOSE. In the DATA DIVISION, the procedural portion has a large data item to facilitate movement of the different data structures from the legacy program to the object-oriented portion of the wrapper. Also in the DATA DIVISION is a table to track subclass names and OBJECT REFERENCES. The code for the procedural portion of the combination wrapper is shown in Figure 7.

How does this combination wrapper affect the legacy program? In the example, all input and output statements must be changed to CALL statements. In the first program, the statement

```
open input input-file.
```

changes to

```
call 'ProgOne'
  using by content z'ProgOneIn'.
call 'Open'
  using by content z'ProgOneIn'
        by reference status-flag.
```

Figure 6 Combination wrapper code—object-oriented portion (part 3 of 5)

```
 Identification Division.
*  Open action
 Method-Id. 'Open'.
 Data Division.
 Local-Storage Section.
 01  file-name pic x(20) value spaces.
 Linkage Section.
 01  file-flag pic x.
 Procedure Division returning file-flag.
*
*  Create instance of object
     invoke CDROM 'somNew' returning fileObj.
*
*  Establish parameter for object-oriented program
     move z'FLIGHT-FILE' to file-name.
*
*  Invoke method
     invoke fileObj 'OpenInputFile' using by content file-name
                                    returning file-flag.
*
*  Return to procedural portion
     exit method.
 End Method 'Open'.

 Identification Division.
*  Close action
 Method-Id. 'Close'.
 Data Division.
 Local-Storage Section.
 01  file-name pic x(20) value spaces.
 Linkage Section.
 01  file-flag pic x.
 Procedure Division returning file-flag.
*
*  Establish parameter for object-oriented program
     move z'FLIGHT-FILE' to file-name.
*
*  Invoke method
     invoke fileObj 'CloseFile' using by content file-name
                                returning file-flag.
*
*  Free instance of object
     invoke fileObj 'somFree'.
*
*  Return to procedural portion
     exit method.
 End Method 'Close'.

 End Class ProgOneInput.
```

and the statement

```
read input-file into input-record
  at end set eof to true.
```

changes to

```
call 'Read-Seq'
  using by content z'ProgOneIn'
        by reference input-record eof-flag.
```

also the statement

```
close input-file.
```

changes to

```
call 'Close'
  using by content z'ProgOneIn'
        by reference status-flag.
```

Figure 6 Combination wrapper code—object-oriented portion (part 4 of 5)

```
 Identification Division.
*  Concrete child class for Program Two's output
 Class-Id. ProgTwoOutput inherits Wrapper3ObjectOriented.
 Environment Division.
 Configuration Section.
 Repository.
     Class CDROM is 'CDROM'
     Class ProgTwoOutput is 'ProgTwoOutput'
     Class Wrapper3ObjectOriented is 'Wrapper3ObjectOriented'.
 Data Division.
 Working-Storage Section.
 01  fileObj object reference.
 Procedure Division.

 Identification Division.
*  Write action
 Method-Id.  'Write'.
 Data Division.
 Local-Storage Section.
 01  file-name pic x(20) value spaces.
 Linkage Section.
*
*  Output record structure
 01  file-record pic x(80).
*
*  Data items passed between legacy and object-oriented programs
 01  file-ptr pointer.
 01  file-flag pic x.
     88 eof value '1'.
 Procedure Division using file-ptr returning file-flag.
*
*  Set address for the output data structure
     set address of file-record to file-ptr.
*
*  Establish parameter for object-oriented program
     move z'PRINT-FILE' to file-name.
*
*  Invoke method
     invoke fileObj 'WriteSeqFile'
          using by content file-name
                by reference file-record
          returning file-flag.
*
*  Return to procedural portion
     exit method.
 End Method 'Write'.
```

Similar changes must be made to all the input and output statements in the second program.

Conclusion

To convert a COBOL legacy program to an object-oriented program, a complete restructuring of the legacy program is required. Objects and their inheritance structure must be identified, data usage and data flow must be analyzed, and instructions allocated to objects. The high costs and risks of this transition are too much for many organizations at a time when budgets are tight. Thus, interoperable legacy and object-oriented applications are highly desirable.

An object-oriented wrapper is useful when a COBOL legacy program is going to become either the server in a client/server system or another object in an object-oriented system. This wrapper encapsulates a COBOL legacy application, transforming its functional interface to an object interface and acting as a "front end" to the legacy application. The object-oriented wrapper is not useful when an object-oriented enhancement is added to the COBOL legacy applica-

Figure 6 Combination wrapper code—object-oriented portion (part 5 of 5)

```
 Identification Division.
*  Open action
 Method-Id. 'Open'.
 Data Division.
 Local-Storage Section.
 01  file-name pic x(20) value spaces.
 Linkage Section.
 01  file-flag pic x.
 Procedure Division returning file-flag.
*
*  Create instance of object
     invoke CDROM 'somNew' returning fileObj.
*
*  Establish parameter for object-oriented program
     move z'PRINT-FILE' to file-name.
*
*  Invoke method
     invoke fileObj 'OpenOutputFile' using by content file-name
                                     returning file-flag.
*
*  Return to procedural portion
     exit method.
 End Method 'Open'.

 Identification Division.
*  Close action
 Method-Id. 'Close'.
 Data Division.
 Local-Storage Section.
 01  file-name pic x(20) value spaces.
 Linkage Section.
 01  file-flag pic x.
 Procedure Division returning file-flag.
*
*  Establish parameter for object-oriented program
     move z'PRINT-FILE' to file-name.
*
*  Invoke method
     invoke fileObj 'CloseFile' using by content file-name
                                returning file-flag.
*
*  Free instance of object
     invoke fileObj 'somFree'.
*
*  Return to procedural portion
     exit method.
 End Method 'Close'.

 End Class ProgTwoOutput.
```

tion and the legacy application needs to use the enhancement in the same way it would use the services provided by a subroutine, because an object-oriented wrapper is itself an object.

The procedural wrapper is useful when a COBOL legacy program must be the main, or client, program and use services provided by an object-oriented program. It is effective if the number of methods in the object-oriented program is known and the number of data items shared between the legacy program and the object-oriented program is small.

The combination wrapper is useful when a COBOL legacy program must be the client program, using services provided by an object-oriented program, but the problem is too complex for a procedural wrapper. The object portion of the combination wrapper is a hierarchy of object wrappers that allow users to take advantage of inheritance to simplify the wrap-

Figure 7 Combination wrapper code—procedural portion (part 1 of 3)

```
Identification Division.
Program-Id. 'Wrapper3-Procedural'.
Environment Division.
Configuration Section.
Repository.
    Class ProgOneInput is 'ProgOneInput'
    Class ProgTwoOutput is 'ProgTwoOutput'.
Data Division.
Working-Storage Section.
*
*  Large table for subclass names and their object references
01  class-table.
    02  class-entry occurs 100 times
                   indexed by ndx.
        03  class-name pic x(30) value spaces.
        03  class-obj object reference value null.
*
*  Other necessary data items
01  anObj object reference value null.
01  aPtr pointer value null.
01  len pic 9(9) binary value 0.
Linkage section.
*
*  Data items passed between legacy and object-oriented programs
01  data-name  pic x(30).
01  data-area  pic x(1000).
01  data-flag  pic x(1).
Procedure Division.
*
*  Program entry point - Program One
    entry 'ProgOne' using data-name.
*
*  Parse name of subclass
    move 0 to len.
    inspect data-name
        tallying len
        for characters before X'00'.
*
*  Find empty place in table
    perform varying ndx from 1 by 1
        until class-name(ndx) = spaces
    end-perform.
*
*  Put subclass name in table
    move data-name(1:len) to class-name(ndx).
*
*  Create instance of subclass and put its object reference in table
    invoke ProgOneInput 'somNew' returning class-obj(ndx).
*
*  Return to legacy program
    exit program.
```

ping process. Existing legacy programs can reuse object-oriented programs by calling the appropriate entry point in the procedural portion, which creates the appropriate objects in the object-oriented portion.

These three wrappers provide viable solutions to the COBOL legacy and object-oriented application interoperability problem for several reasons.

Minimal changes are required to the legacy code. Without a wrapper, for a legacy program to interact with an object-oriented program the *long names* option, which affects all the subprogram and entry point names in the program, is required. A REPOSITORY paragraph must be added in the ENVIRONMENT DIVISION and OBJECT REFERENCE data items must be added in the DATA DIVISION. The PROCEDURE

Figure 7 Combination wrapper code—procedural portion (part 2 of 3)

```
*  Program entry point - Program Two
     entry 'ProgTwo' using data-name.
     move 0 to len.
     inspect data-name
         tallying len
         for characters before X'00'.
     perform varying ndx from 1 by 1
         until class-name(ndx) = spaces
     end-perform.
     move data-name(1:len) to class-name(ndx).
     invoke ProgTwoOutput 'somNew' returning class-obj(ndx).
     exit program.
*
*
*  Action entry point - Read Sequential
     entry 'ReadSeq' using data-name data-area data-flag.
*
*  Set pointer to input data structure
     set aPtr to address of data-area.
*
*  Parse name of subclass
     move 0 to len.
     inspect data-name
         tallying len
         for characters before X'00'.
*
*  Find subclass name in table
     perform varying ndx from 1 by 1
         until class-name(ndx) = data-name(1:len)
     end-perform.
*
*  Invoke method
     invoke class-obj(ndx) 'Read' using aPtr
                                  returning data-flag.
*
*  Return to legacy program
     exit program.
*
*
*  Action entry point - Write Sequential
     entry 'WriteSeq' using data-name data-area data-flag.
     set aPtr to address of data-area.
     move 0 to len.
     inspect data-name
         tallying len
         for characters before X'00'.
     perform varying ndx from 1 by 1
         until class-name(ndx) = data-name(1:len)
     end-perform.
     invoke class-obj(ndx) 'Write' using aPtr
                                   returning data-flag.
     exit program.
```

DIVISION must be rewritten to create object instances and INVOKE methods, a monumental task.

With a wrapper, a few selected statements in the PROCEDURE DIVISION are replaced or removed. In some cases, minor changes are needed in the DATA DIVISION. If the wrapper and the object-oriented program have already been debugged and tested with a stub program, it is easy to test the new application and quickly get it into production.

New features can be written using object-oriented technology. This provides all the benefits of object-oriented technology, allowing fast response to system development needs. Thus higher quality software for more complex data types and problem domains can be developed.

The object-oriented code is easier to maintain and changes do not affect the legacy program.[5] The object-oriented code does not have to be written in

Figure 7 Combination wrapper code—procedural portion (part 3 of 3)

```
*  Action entry point - Open
     entry 'Open' using data-name data-flag.
     move 0 to len.
     inspect data-name
         tallying len
         for characters before X'00'.
     perform varying ndx from 1 by 1
         until class-name(ndx) = data-name(1:len)
     end-perform.
     invoke class-obj(ndx) 'Open' returning data-flag.
     exit program.
*
*
*  Action entry point - Close
     entry 'Close' using data-name data-flag.
     perform varying ndx from 1 by 1
         until class-name(ndx) = data-name(1:len)
     end-perform.
     invoke class-obj(ndx) 'Close' returning data-flag.
*
*  Free the instance and clean up table entry
     invoke class-obj(ndx) 'somFree'.
     set class-obj(ndx) to null.
     move spaces to class-name(ndx).
     exit program.

End Program 'Wrapper3-Procedural'.
```

COBOL. If the new feature is easier to write and maintain in another object-oriented language, the wrappers presented in this paper can still be used.

Object-oriented programs can reuse or be reused by any number of legacy programs. An object wrapper provides a way to reuse the functionality of a legacy program as a black box, without duplicating the functions for every new application. Further, the object wrapper ensures that the client program can handle different legacy programs with the same user interface standards.

Procedural and combination wrappers provide a way for a legacy program to reuse object-oriented program functionality, again with no duplication. These two wrappers are building blocks that provide a binding of data and function and enable the appropriate object creation and method invocation.

Since reuse is one of the biggest benefits of object-oriented technology, care should be taken to make the wrapper extensible. Imagine that a wrapper might be used eventually by every legacy program. Although a gross generalization, this mind-set helps the wrapper to be easily modified when another legacy program does want to use it.

Wrappers allow legacy applications to find their way into new applications that were once technically difficult or economically infeasible. Making legacy and object-oriented applications work together is like a jigsaw puzzle—the connecting piece that completes the picture is not always easy to find. The suggestions in this paper may reveal a missing piece.

Acknowledgments

Many people at the Santa Teresa Lab worked very hard to make the new object-oriented COBOL compilers a reality. I want to acknowledge Tom Dunham, who made my work with object-oriented COBOL possible, and Connie Nelin and Steve Miller, who taught me well.

Cited references and notes

1. W. Dietrich, L. Nackman, and F. Gracer, "Saving a Legacy with Objects," *OOPSLA '89 Conference Proceedings*, New Orleans, LA, ACM Press (1989), pp. 77–83.
2. Gartner Group, 1996.
3. A. W. Hui, "An Object Wrapper for a Legacy System," presented at the 1995 International Conference on Object Technology, available from the author (alanhui@raleigh.ibm.com).
4. Since the wrapper is an object-oriented program, the program that creates an instance of the GUI object must also create an instance of the wrapper object.
5. It is important to note, however, that a signature change in a method in the object-oriented code does require a change in the wrapper.

General references

E. C. Arranga and F. P. Coyle, "Object COBOL," *Object Magazine* **4**, No. 5, 56–62 (September 1994).

E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley Publishing Co., Reading, MA (1995).

L. Hellenack, "IBM Object-Oriented COBOL: It's Ready for Prime Time," Report No. 10100, International Data Corporation, Framingham, MA (July 1995).

A. V. Hense, "Denotational Semantics of an Object-Oriented Programming Language with Explicit Wrappers," *Format Aspects of Computing* **5**, No. 3, 181–207 (1993).

T. W. Miller, G. Miller, and M. P. Nally, "Interfacing Between Structured and Object-Oriented Languages," IBM Technical Disclosure Bulletin **37**, No. 3 (March 1994).

Accepted for publication July 2, 1996.

Elizabeth Spangler Flint *IBM Software Solutions Division, Santa Teresa Laboratory, 555 Bailey Avenue, San Jose, California 95161-9023 (electronic mail: esflint@vnet.ibm.com).*
 Ms. Flint has been an IBM employee for the past ten years with work assignments in Cary and Charlotte, North Carolina, and with IBM Education and Training. Currently, Ms. Flint is a member of the Year 2000 Technical Support Center at the Santa Teresa Laboratory, although she works from her home in Union, South Carolina. Ms. Flint earned her B.S. degree in mathematics and computer science from Virginia Polytechnic Institute and State University and her M.S. degree in computer science from North Carolina State University.

Reprint Order No. G321-5634.