Object-oriented COBOL recycling

Conference Paper · December 0196

DOI: 10.1109/WCRE.1996.558901 · Source: IEEE Xplore

Harry Sneed
ANECON GmbH
220 PUBLICATIONS **2,621** CITATIONS

Object-Oriented COBOL Recycling

by
Harry M. Sneed
SES Software-Engineering Service GmbH
Rosenheimer Landstrasse 37
D-85521 Ottobrunn, Germany
Fax ++49-(0)89-60853671
Email: 100276.57@compuserve.com

Abstract

In this paper a tool supported process for extracting objects from existing COBOL programs is described. The process is based on human interaction to select objects coupled with automated slicing techniques to identify all of the elementary operations which change the state of the object selected. The object is redefined within the framework of an Object-COBOL class and the elementary operations are attached to it as methods. The result is a set of Object-COBOL classes.

1 The Rationale for Object-Oriented Reengineering

Object-oriented reengineering has the goal of transforming existing procedural systems into an object-oriented architecture. The procedural systems may be structured or not, if not they should first go through structural reengineering. The result of object-oriented reengineering should, in any case, be a set of objects which perform the same functions as the previous procedures.

Object technology is definitely the predominant software trend of the 1990's. Whether it will fulfill all of the expectations is another matter. According to the literature it should enhance maintainability, reduce the error rate, increase productivity and make the data processing world a better place to live.[1] Of course, as with all new technologies there is a lot of marketing hype connected to it. However, for distributed applications with graphical user interfaces, there seems to be no alternative. Object orientation is a necessary precondition to realizing complex networked systems. The OMG's CORBA - Common Object Request Broker Architecture - standard is well on its way to becoming a world wide standard for accessing data and objects in a distributed computer network and for exchanging messages between objects on different computers. The key to CORBA is the IDL - Interface Definition Language - for specifying the interfaces.[2] Through CORBA it is possible to even access legacy code on a mainframe to provide services for the clients on the periphery. This technique, known as wrapping, is an alternative to object-oriented Reengineering.[3]

The ability to reuse code is certainly enhanced through object-oriented programming and in particular through object-oriented architectures. There is also reason to believe that the object-oriented systems are more flexible and adaptable, but the claims about improved productivity and maintainability still remain to be verified. Important is the emphasize given by the system suppliers. All the new software architectures, be it DOE - Distributed Objects Everywhere - from SUN, Object Broker from DEC, ORBIX from IONA or SOM - Systems Object Modelling - from IBM, are based on object technology. Thus, in spite of the scepticism of many critics, the shift to the new paradigm is inevitable. For better or for worse, object-orientation is the target software technology of the 90's.

The greatest obstacle to the introduction of object technology is the achievement of past technologies, i.e. the presence of so many legacy systems and the people who maintain them. As long as these systems are operating sufficiently, there is no pressing need to replace them, even if the new systems promise to be better. They would really have to be significantly better, in order to take the risk of introducing them. Wise managers have learned from past experience how difficult it is to develop new software systems. The costs of development are difficult to predict, particularly with a new technology such as object orientation. There is also a high risk that the project will fail - according to the literature on risk assessment at least 40 %. So why should anyone in his right mind want to redevelop a working system just for the sake of being object oriented.[4]

The redevelopment of existing systems is only justifiable in those cases where the existing procedural oriented systems are either obsolete or unmaintainable. As long as these systems are functionally acceptable there is no real need to replace them. This is true for most backend applications today performing batch jobs on mass databases. It is also true for some online teleprocessing applications performing fixed transactions on large databases, but most online applications are outdated. Not only are their full screen panels obsolete, their whole transaction handling philosophy needs to be updated. Thus, although the functionality is basically acceptable, there is a pressing need to port them onto a new computer platform. This need will always arise when the old hardware is no longer sufficient or when significant benefits are to be gained from the new software environment - benefits such as greater flexibility, better efficiency, more compatibility, and enhanced interoperability.

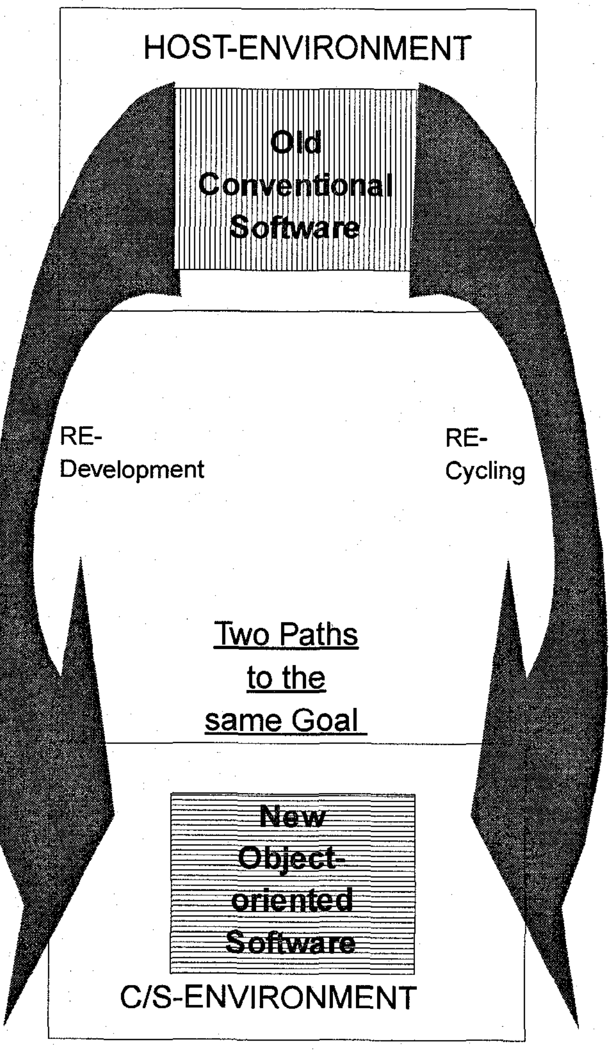
HOST-ENVIRONMENT
Old Conventional Software
RE-Development
RE-Cycling
Two Paths to the same Goal
New Object-oriented Software
C/S-ENVIRONMENT

Fig. 0: Object-oriented Reengineering

There is, of course the alternative of wrapping the old applications and communicating with them via a standard application program interface. However, this solution is not always feasible. The old software may be so poorly constructed that it cannot be maintained as it is. It may also be a strategy of the user to leave the old environment altogether, such as the case when downsizing from the host to a distributed system. In this case, the user will want to migrate as much of the old code as possible into the new environment. Finally, even when developing new applications for a new platform, it may be economical to extract portions of the old code for reuse in the new programs.

In summary, the major reasons for object-oriented reengineering are

- to take the advantage of the many possibilities offered by object technology,
- to free oneself from a proprietary environment,
- to reap the benefits of an open distributed system environment,
- to reduce maintenance costs, and
- to be able to reuse old programs and data as objects in a new system architecture.

(See figure 0).

2 Obstacles to Object-Oriented Reengineering

Having established a need for object-oriented reengineering, the next step is to define an appropriate reengineering process. The technical problem is how to restructure existing programs so that certain data structures can be converted to abstract data types and the elementary operations of the program can be assigned to them. The result should be a hierarchy of classes, each with an encapsulated data object and a number of methods for processing that data object which can be accessed from outside. The subordinate classes should inherit features, i.e. data structures and operations, from the superordinate classes of the class hierarchy. Each class should also be reusable in another context other than the one from which it is derived. Such is the problem to be solved. There are unfortunately many obstacles to be overcome in solving this problem.

Most legacy programs are procedural in nature. They are constructed to fulfill a given application function such as processing order entries. In order to fulfill their function they have to access a number of different data sets. A typical application program accepts a transaction from a terminal or an input file, checks that transaction against one or more master files or databases, updates several master files and databases, produces some new files and prints a protocol report. The action on the data are carried out in a given procedural order and this is exactly the order in which the elementary operations of the program are grouped. The program is a mirror not of the data but of the procedure. Thus, the elementary operations, or code blocks, are grouped together on the basis of their execution sequence regardless of the objects they process.

This chain of operations is triggered by invoking the initial operation which in turn passes control onto the others. The others are reached by GOTO branches, by PERFORM's or subroutine calls or by simply falling through. Therefore, although the operations of an existing procedural program may be potential methods, it can be extremely difficult to disentangle them from one another. This is the first obstacle to be overcome.

The problem is further compounded by the fact that any one block of code may process many data objects. Often in assignment statements, data is moved from one object to another, such as from the panel to a database view and vice versa. Thus, if object methods are to be derived from existing procedural code blocks, the latter will have to be sliced up by object, creating a partial procedure for each object processed. This is the case for data flow slicing.[5] The result could easily be methods of only one or two statements. If the execution sequence is to be preserved there will be no other choice but to leave them in that state, meaning there will be a plethora of atomic methods. This is the second problem to be solved.

A third problem is that of the object definition itself. It may at first glance appear that the data objects of a program are obvious, namely, the files, views, reports and panels processed. However, a second glance shows that it is not quite so simple as that. These external data structures are often transferred to internal data structures and processed there. These internal data structures are copies of the external ones and have to made equivalent to them when defining data objects. Besides there is no automatic rule for object definition. Objects can only be defined in the context with which they are used. That means application knowledge is required, if the object selection is to be meaningful.

A fourth obstacle to be overcome is that of redundant code. Since the legacy programs are structured by execution sequence, the same operations may occur many times within the same program as well as in different programs. If the code is reordered by data object so that each elementary operation is assigned to the object whose state it changes, there will be many occurrences of the same operation, i.e. method, within a given class. If the derived classes are to be maintained, it will be necessary to recognize and remove such redundant methods.

A final problem is that of the naming practices. In many legacy systems, procedures and data are named arbitrarily. Programmers often choose to name procedures after their girlfriends or favorite sportsmen, so that the names have no relation to the actual function. Besides that, totally different procedures may have the same name in different programs, whereas identical procedures may have different names. The same applies to data items. Unless the user has used standard COPY or INCLUDE members, data attributes of the same structure may have different names from one program to another. Even worse is the case when the same name is used for quite different data objects. These synonym and homonym problems have to be resolved if the reengineering action is going to be successful.

Taken in their totality, the obstacles to object-oriented reengineering are much greater than those of classical procedural reengineering. This is due to the tremendously wide gap between procedural and object-oriented architecture. Users must be aware of this architectural mismatch as pointed out by Garban, Allen and Ockerbloom in their landmark paper „Architectual Mismatch- Why it is so hard to build systems out of existing parts".[6] Bridging the gap will require a significant effort no matter how it is done.

3 Prerequisites for Object-Oriented Reengineering

Transforming programs from a procedural to an object-oriented structure presupposes that the programs are structured and modular, otherwise they cannot be transformed. Structured means that there are no GOTO like branches from one segment of code - in COBOL paragraphs, in PL/I internal procedures and in FORTRAN subroutines - to another. There may be GOTOS within a segment of code but not outside of it. Modular means that the programs are segmented in a hierarchy of code segments, each with a single entry and a single exit. The segments or procedures should correspond to the elementary operations of the program. Each operation should be reachable by invoking it from a higher level routine. These two prerequisites - modularity and structuredness - are achieved by procedural reengineering as described earlier.[7] The third prerequisite is to establish a procedural calling tree for the subroutine hierarchy so that all subroutines are included as part of the procedure which calls or performs them. This has to be done to ensure that subroutines are handled as an extension of the main routine when doing data flow analysis.

Thus the three major prerequisites for object-oriented reengineering are

- procedural structuring,
- modularization, and
- inclusion of subroutines.

In the case of COBOL this means that the program is a set of procedural paragraphs which are performed from one central control paragraph, that there are no GOTO's between paragraphs and that PERFORMED paragraphs are considered to be part of the paragraph which performs them. The program to be converted now has a single global data area - the DATA DIVISION and a single procedural code sequence - the PROCEDURE DIVISION.

4 The Object-Oriented Reengineering Process

Object-oriented reengineering is a five step process with the steps

- object selection,
- operation extraction,
- feature inheritance,
- redundancy elimination
- syntax conversion

(see figure 1)

4.1 Selecting the Objects

The first step in the object-oriented reengineering process itself is to identify the data objects. There have been some attempts within the research community to try and identify objects automatically. Such an attempt was made in the ESPRIT REDO project and reported on by Lano and Breuer.[8] These attempts are academically interesting but practically unusable. It is impossible to judge what might be a good potential object. Lano suggested that objects could be the records that a program processes. This author has extended the notion of a record to include IMS segments, SQL views, panels or maps, report formats and tables, i.e. all of the technical data access units that a program can process. This could be a starting point. However, in principle, anything could form a data object, even a single data item such as a date field. Only the owner of a program who is familiar with the functional domain and who also knows the operational environment can determine what objects should be. Segments, records, views, panels and reports will probably become objects, but so will groups of logically related work data and internal tables. There is no reliable way to determine this automatically. It has to be done by a knowledgeable human being.

So the practical solution is to display the entire DATA DIVISION, i.e. the complete set of data variables of a program and let the user choose his objects. This can be done by having the user mark those data items which he wants to be attributes of a particular object and then submit an object name. The supporting tool will then collect these attributes into a single data structure if they are not already in such a structure and assign the structure the name of the object.

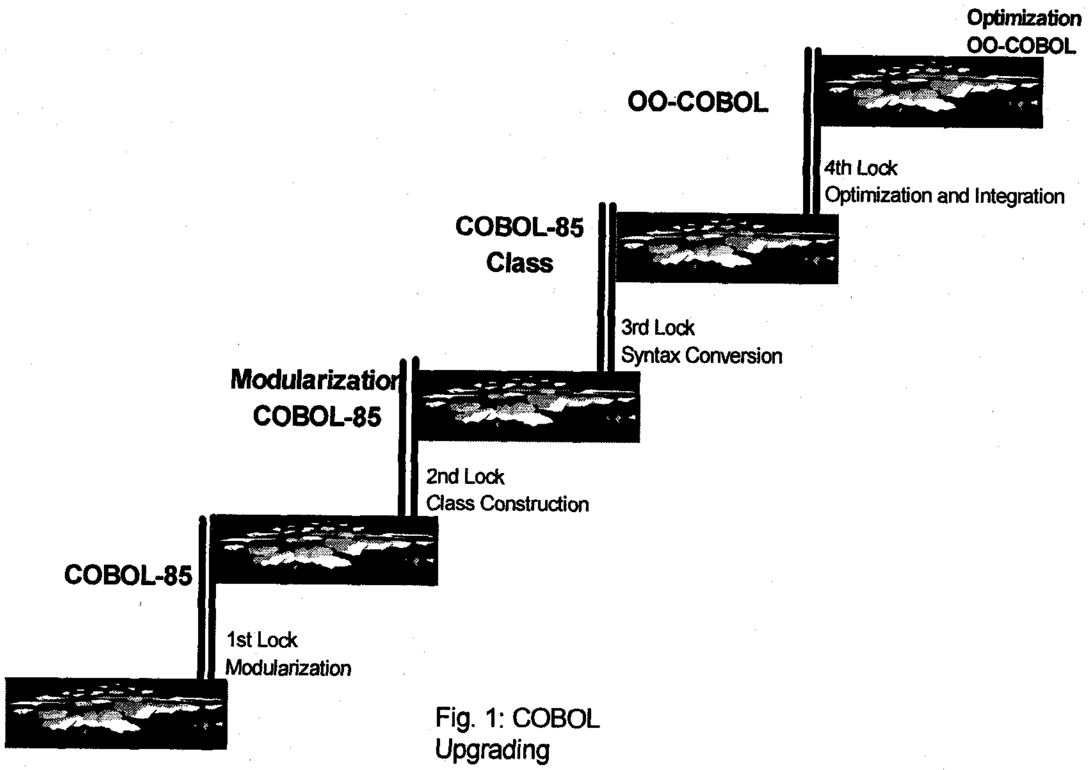

Fig. 1: COBOL Upgrading

This selection action should be repeated for each object the user wants to extract from the original program. The non selected data will remain as variables in the user main program. The data selected will be removed from the main program to form the core of a new reusable class to be used by this main program as well as by all others. Thus, by extracting the account data from a program and assigning it to an object account the user is making it available to all other programs which use the account data. That means, he will also have to remove the account data from the other programs as well. In the end there will be only one class for account data and it will include all the attributes of an account from all of the programs. The final class account is a cartesian product of all account fields.

The object selection must be made by the user, but it can be tool supported. The tool can aide the user in selecting data, it can assemble the data into a single structure and it can check if the data items selected in one program correspond to the data items selected in another. If the names are the same, the data attributes such as type, length and occurrence factor should also be the same. If not the user should get a warning.

As a result of the object selection step, there will be a set of data objects each with a data structure including at least two variables - an object identifier or key and an object attribute. As a rule an object will have many attributes such as a customer record with:

- CUSTOMER-NO
- CUSTOMER-NAME
- CUSTOMER-TEL-NO
- CUSTOMER-ADDRESS
- CUSTOMER-CREDIT-LIMIT
- CUSTOMER-CREDIT
- CUSTOMER-DEBIT
- etc.

On the other hand, an object should not have too many attributes, otherwise the class will include too many operations and become too big to handle. This is where the object selector has to exercise good judgement. (see figure 2).

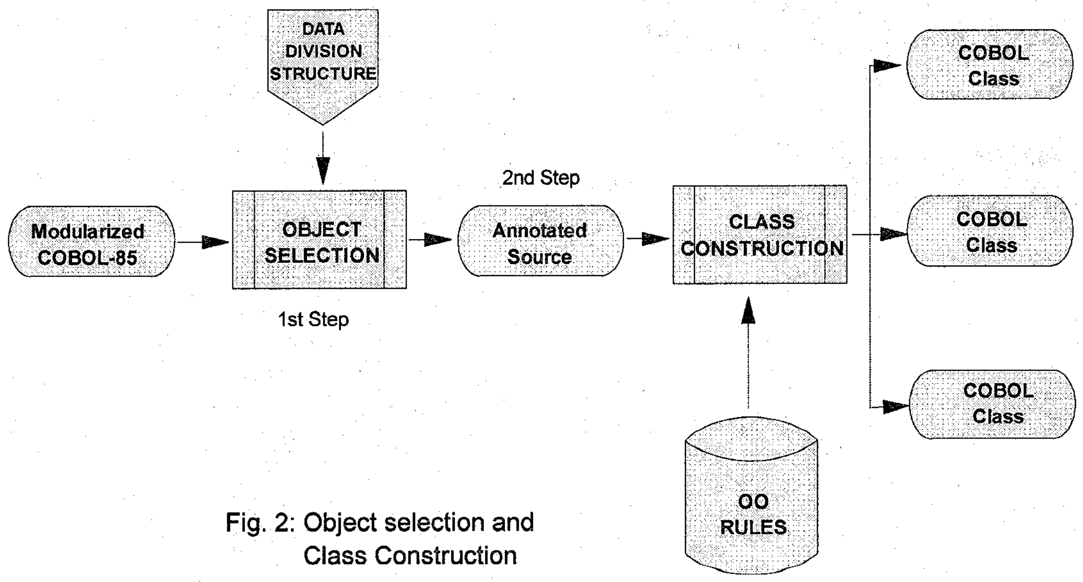

Fig. 2: Object selection and Class Construction

4.2 Extracting the operations

The second step in the object-oriented reengineering process is to extract all of the operations performed upon the selected objects. The procedural part of the original program consists of control commands, assignments, computations and input/output operations. The assignments and computations set or change the state of a particular data variable. The input/output operations transport data groups in and out of the address space of the program.

In an object-oriented program architecture the operations upon an object, i.e. all statements which alter the state of the object, as well as all input/output operations for that object, should be encapsulated together with the object data. The objective here is to create an abstract data module for each object selected by the user which includes the statements performed on each object. This entails parsing the procedural code and extracting all statements in which an attribute of the selected object is assigned or computed and all i/o statements which transport that object data.

Thus, if the object selected is the customer data, then the statements

```
READ CUSTOMER
MOVE ZERO TO CUSTOMER-CREDIT
ADD PURCHASE-AMOUNT TO CUSTOMER-
DEBIT
STRING NEW-NAME DELIMITED BY SIZE
        NEW-STREET DELIMITED BY SIZE
        NEW-CITY DELIMITED BY SIZE
        NEW-ZIP DELIMITED BY SIZE
INTO CUSTOMER-ADDRESS
END-STRING
REWRITE CUSTOMER
```

should be extracted from the main program and assigned to the object customer.

The statements do not have to be contiguous. They may be scattered throughout the source. If two or more consecutive statements act upon the same object, they form a compound operation. Single statements and compound operations will form the methods of the target object. They are assigned the name of the procedure to which they originally belonged plus a unique number to identify them and are inserted into a source member where the object data is declared. Now the object data and the object operations are together in one module.

```
*OBJECT = CUSTOMER.
  01 CUSTOMER.
     05 CUSTOMER-NO              PIC 9(8).
     05 CUSTOMER-NAME  PIC X(30).
     05 CUSTOMER-TEL-NOPIC X(10).
     05 CUSTOMER-ADDRESS.
        10 CUSTOMER-ADR-NAME      PIC X(30).
        10 CUSTOMER-ADR-STREET    PIC X(40).
        10 CUSTOMER-ADR-CITY      PIC X(20).
        10 CUSTOMER-ADR-ZIP       PIC 9(5).
     05 CUSTOMER-CREDIT-LIMIT     PIC S9(5)v99.
     05 CUSTOMER-CREDITPIC S9(5)v99.
     05 CUSTOMER-DEBIT  PIC S9(5)v99.

PROCESS-CUSTOMER-1.
       READ CUSTOMER
              INVALID KEY
              MOVE READ-ERROR TO RET-CODE
       END-READ.

PROCESS-CUSTOMER-2.
       MOVE ZERO TO CUSTOMER-CREDIT.
       MOVE PURCHASE-AMOUNT
            TO CUSTOMER-DEBIT.

PROCESS-CUSTOMER-3.
       STRING NEW-NAME  DELIMITED BY SIZE
              NEW-STREET  DELIMITED BY SIZE
              NEW-CITY  DELIMITED BY SIZE
              NEW-ZIP  DELIMITED BY SIZE
          INTO CUSTOMER-ADDRESS
       END-STRING.

PROCESS-CUSTOMER-4.
       REWRITE CUSTOMER.
```

As a result of the operation extraction there will be a source module for each object selected in which the data attributes and the elementary operations are collected.

In the original program which is stripped of the object relevant operations, the operations are replaced by CALL's. Thus the elementary operation

```
MOVE ZERO TO CUSTOMER-CREDIT
MOVE PURCHASE-AMOUNT
     TO CUSTOMER-DEBIT
```

is replaced by the CALL command

```
CALL "CUSTOMER" USING
   "PROCESS-CUSTOMER-2" BY CONTENT
   PURCHASE-AMOUNT,
   RET-CODE.
```

If the user has assigned most of the original program data to objects, then most of the input/output operations, assignments and computations will also be assigned to objects. The procedural part of the original program will remain a skeleton of control commands - selections, repetitions and subroutine invocations. Only the operations on the global data will remain. The code segments removed will have been replaced by CALL´s to their object where the local data is encapsulated.

4.3 Inheriting the extracted features

The third step in the object-oriented reengineering process is for the target program which has now been stripped of the objects which it processes to inherit the features of those objects. All of those data elements which were removed in the object selection step are now declared to be inherited by referring to the object which they belong. Therefore, the data elements themselves need no longer to be declared in the target program, only the object references.

The CUSTOMER record which was extracted to create the CLASS CUSTOMER is now referenced as

```
01  CUSTOMER-DATA   REFERENCE OBJECT.
```

just as the ARTICLE record is now referenced as

```
01  ARTICLE-DATA   REFERENCE OBJECT.
```

In the ENVIRONMENT DIVISION, it has to be noted that these data structures are inherited along with their methods which are invoked. This means that the program has been inverted. It is now subordinate to the objects which have been extracted from it. These objects may also become superordinate classes for other programs as well.

4.4 Merging extracted classes of the same type

It is obvious that a class extracted from an existing program - for instance the class ARTICLE- will also be extracted from other programs as well. Thus in the end, there will be a number of classes for the same object type with the same attributes and the same or similar methods. The idea of an object-oriented architecture is, of course, to have only one class declaration for each object type, i.e. one ARTICLE class.
This means that all ARTICLE classes extracted from the various programs must be merged into a single all encompassing superclass which will include all attributes and all methods. On the other hand, each attribute and each method may only occur once. So all data with the same name is merged into a common field with a common type and length. In doing so it may have to be redefined to allow for incompatible types.

More difficult, is the task of eliminating redundant methods. Of course, if one method has exactly the same statements in the same order as another, it is obvious that they are redundant. It is more difficult to detect redundancies, if the methods have slightly different statements which produce the same result. Here it is necessary to apply pattern matching techniques in order to detect redundancy.[9] In the end, the number of methods should be reduced drastically. In any case there should only be one method for each type of IO database operation.
(See figure 3).

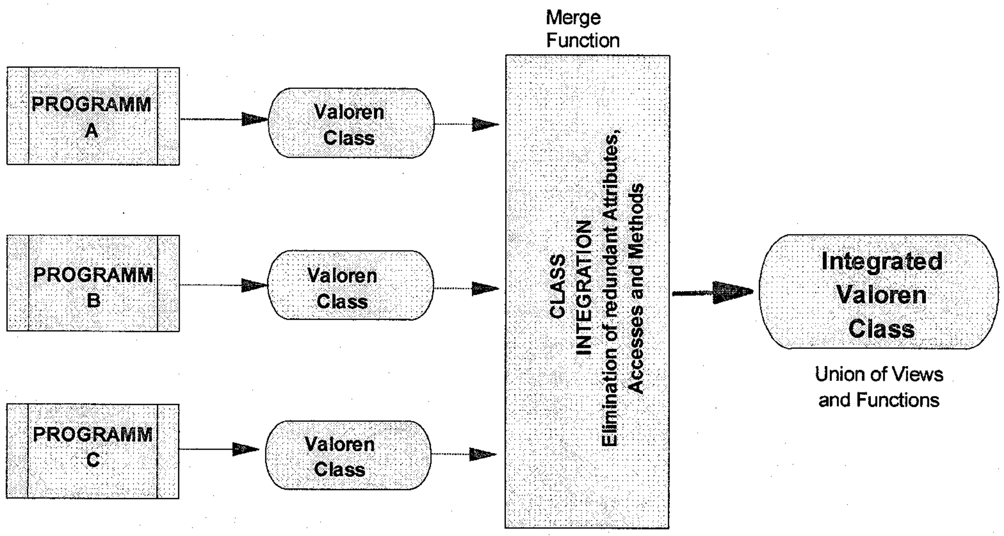

Fig. 3: Class Integration

4.5 Converting the extracted classes to Object-COBOL syntax

The final step after eliminating redundant classes is to convert the remaining classes to Object-COBOL. This is a straight forward conversion process.
Some commands are replaced such as

```
  PROGRAM-ID. CUSTOMER. by
  CLASS-ID. CUSTOMER  INHERITS FROM PERSON.
```

other commands are enhanced such as

```
  01     CUSTOMER. to
  01     CUSTOMER   USAGE OBJECT.

         and

  PROCESS-PURCHASE.  to
  METHOD-ID.   PROCESS-PURCHASE IS PUBLIC.
```

For each method, i.e. former Paragraph, a Linkage Section is inserted which contains its parameters e.g.

```
  LINKAGE SECTION.
  77  PURCHASE-AMOUNT PIC S9(5)v99.
```

an entry statement is inserted at the beginning,

```
PROCEDURE DIVISION USING PURCHASE-AMOUNT
                         RETURNING RET-CODE
```

and an exit statement is inserted at the end

```
  EXIT METHOD
  END-METHOD   PROCESS-PURCHASE.
```

Finally, the DATA DIVISION is partitioned into an Object-Storage Section where the object such as CUSTOMER is declared, a File-Section and a Working-Storage Section where the global variables are declared. Local variables are placed in the Working-Storage Section of each method.

The resulting OO-COBOL class conforms to the current draft standard of the CODAYSL X3J4 Task Group.[10] As such it is compilable by either the Micro-Focus Object-COBOL Compiler under MS-Windows or the IBM Visual-COBOL Compiler under OS/2. However, the reengineered classes are still far from being ready. They need to be checked and corrected by a human expert familiar with Object-COBOL.

The idea of COBOL-Recycling is not to automatically provide a final executable solution but to provide a set of reusable classes extracted from existing host applications which with a little adjustment can be built into new distributed applications, thus saving the effort of rewriting all of the existing functions. The overall control flow of the new client/server applications is going to have to be redesigned and rewritten in any case, but there is no need to rewrite the elementary operations. These can be extracted from the existing code, converted to methods and attached to the objects they process.

5 Future Directions in Object-Oriented Reengineering

It is clear that object-oriented reengineering needs to be further explored and refined in order to be exploited by industry. The transformation of procedural programs into object-oriented ones is not trivial. It is a complex multi-step, m:n transformation process which requires human intervention in deciding what objects to be chosen. The process described here has been developed and tested by the author on individual programs.[11] It now has to be scaled up to handle larger systems with many programs. The underlying theory of program slicing by data usage has proved to be sound, but it remains to be seen how the different slices can interact in a distributed environment.

Real customer driven projects are necessary to validate the approach. Users are only beginning to migrate to object technology and progress is particularly slow in the COBOL world. So it will take some time, before the approach can really be put to a test. In the meantime, it will be perfected.

References

[1] Meyer,B.: *Object-Oriented Software Construction*, Prentice-Hall International, London, 1988, pp. 3-26

[2] Mowbray, T.I., Zahavi,R.: *The Essential CORBA*, Object Management Goup, John Wiley, New York, 1995, p. 41

[3] Winsbery,P.: "*Legacy Code- Don't Reengineer it Wrap it*", in Datamation, May 1995, pp. 36-41

[4] Charette,R.: *Software Engineering Risk Analysis and Management*, McGraw-Hill, New York, 1989, p. 27

[5] Canfora, G., Cimitile, A., Munro, M.: "*RE*^{2}": *Reverse-Engineering and Reuse Reengineering*" Journal of Software Maintenance, Vol.6., No.2, 1994, p. 53

[6] Carbn, D., Allen, R., Ockerbloom,I.: "*Architectural Mismatch - Why it´s so hard to build systems out of existing parts*" in IEEE Software, Nov. 1995, pp. 17-26

[7] Sneed, H., Jandrasics,G.: "*Software Recycling*" in Proc. of CSM, IEEE Press, Austin Tx, 1987, pp. 82-90

[8] Lano, K., Brever,P.: "*Reverse Engineering COBOL via Formal Methods*" in the REDO Compendium, editor: H.J. van Zuylen, Esprit Report, John Wiley, Chichester, 1993, pp. 229-248

[9] Engberts, A., Kozaczynski, W., Ning.: "*Concept Recognition-Based Program Transformation*" in Proc. of CSM, IEEE Press, Sorrento, 1991, pp. 73-82

[10] Obin, R.: *Object Orientation - An Introduction for COBOL Programmers*, MicroFocus Press, Palo Alto, CA., 1993

[11] Sneed, H.: "*Migration of Procedurally oriented COBOL Programs in an Object-Oriented Architecture*" in Proc. of CSM, IEEE Press, Orlando, 1992, pp. 109-116