Migration of Procedurally Oriented COBOL Programs in an Object-Oriented Architecture

Harry M. Sneed

Software Engineering Service
Rosenheimer Landstr. 37, D-8012 Ottobrunn, Germany

The subject of this paper is the migration of procedurally structured COBOL programs into functionally equivalent object-oriented programs. The major differences in procedural and object-oriented program architecture are described and an approach presented to bridge the gap between the two. The approach is based on the partitioning of programs into abstract data types and the reallocation of procedural code based on objects of processing. The result is a set of classes corresponding to the proposed standard for object-oriented COBOL.

1. The Need for a New Software Paradigm

The last major paradigm change occurred in the 1970's, when most commercial data processing shops switched from batch to online processing. Several thousand man-years went into reprogramming applications to become transaction-oriented. If they were not totally rewritten, the programs had to be at least totally reorganized. Instead of being driven by input files the programs were made to be driven by the superordinate transaction monitor. Main programs with a nested loop structure became subprograms with a case type structure. A great deal of new code had to be added just for housekeeping purposes, error handling, restarting, communication with other programs, etc.

Now, the data processing community is faced with a new transition - that to distributed processing in a client-server architecture. For this, the application software should again be redeveloped, this time in accordance with the object-oriented paradigm. However, those who went through the pain of moving from batch to online are hesitant. It is a widely accepted fact that moving from an action-oriented architecture into an object-oriented one is a major undertaking with a high risk.[1] The costs of this transition promise to be much greater than the costs of the last one and this at a time when budgets are tight. Therefore, many shops, including the Union Bank of Switzerland, are looking to automated reengineering techniques to help reduce the effort in migrating the programs from one environment to the other. The objective is to retain as much of the old code as possible, while at the same time reorganizing the programs and adding new built-in features. That is the main motivation for the research work being described here.

2. Characteristics of Conventional COBOL Programs

Conventional COBOL programs belong to one of three classes:

-- batch, online or subprograms.[2]

Batch programs are constructed to update databases or master files from one or more transaction files and to
produce reports on information obtained from a database or master files. In the updating case, the database records are the object of the processing, however, the program is structured around the updating actions — delete, insert, and modify. Each updating action uses one or more incoming transaction records to alter one or more master records.

In the report generation case, the conventional program structure reflects the structure of the report, i.e., the code grouping corresponds to the output grouping. The database is queried to obtain the data required and this is sorted and reordered to match the structure of the report. Considering the report as an object, the report generating program is closer to the object-oriented philosophy than the update program except for the input side which processes a mixture of different objects. In general, it can be stated that large batch programs with their diverse inputs and outputs, and their dataflow-oriented design are the most far removed from an object-oriented approach.

Online transaction processing programs are built around the transactions which they process and these are reflected in the user interface panels or maps which make up the transaction. Simple transactions have only one map. Complex transactions may have a whole string of interrelated maps. The map content determines the action to be taken by the program.

Online updates may access several files from one panel to alter, insert, or delete data. In this case, there is a one to many relationships between the source of the action and the objects of the action. Query programs produce a report map from one or more source database files. So here the relationship is a many to one.

In most online applications, there is a one to one relationship between the user interface panel and a program, i.e., one program processes one map. Considering the map to be an object, this makes the online transaction processing programs a little closer to an object-oriented architecture than the batch ones. The main difference is the combined access to multiple database objects within the same module. This causes some modules to be unusually large. When different maps are combined into one module, the complexity increases even more.

Subprograms normally are invoked to provide a particular function too large or too complex to be included in the main program. They are also used to implement general purpose functions required by many main programs, in which case, they embody reusable code. The object of the subroutine actions is a set of parameters passed at runtime. In many cases, the subroutine will alter some parameter values based on the values of others. If the parameter list is considered an object, the subprogram is already object-oriented. In other cases, where the subroutine accesses files or prints reports, the parameters can be looked upon as a message triggered some action on some object. Thus, the COBOL subprogram is usually closer to the object-oriented philosophy than either of the two main program types.

3. Characteristics of Object-Oriented COBOL Programs

Having examined the characteristics of existing COBOL applications, the next step to defining the requirements for a reengineering process is to look at the characteristics object-oriented COBOL applications should have. These characteristics have been proposed by the ANSI COBOL Committee and enhanced
by Yourdon, Microfocus, and others.[4]

First, programs should be divided up into classes corresponding to the objects processed. Each class is an abstract data type encapsulating the attributes of and the actions on the object enclosed. The criteria for partitioning the programs are the abstract data types or objects processed. There is one module per object.

Secondly, classes are isolated against outside access. No external class can access data within the base class, except for those subordinate to it with inheritance rights. Classes communicate with one another via message sending. The messages are declared in a separate import/export area and are passed as parameters in a CALL statement. In this way, information hiding is enforced.

Thirdly, classes should be designed so as not to exceed a certain size and complexity. No objects should have more than 100 attributes nor more than ten methods. A method should not contain more than 20 statements. A message should be restricted to five parameters. These constraints are intended to insure a high degree of modularity.[5]

Fourthly, classes should be able to inherit data attributes from more than one superordinate class, i.e. they have multiple inheritance. The inherited data items should be explicitly declared with reference to the class from which they are derived. The same applies to methods which are inherited from superordinate classes.

Fifthly, classes should be able to communicate with one another through message passing. Any one class can communicate with any other external class by calling it with a list of parameters. Each method of a class is defined as an entry point with a definition of the parameters it receives, so it can be invoked directly. Control can be passed back to the caller or on to another class. Subordinate classes can be invoked without parameters, as they have access to the data of the invoking class.

Sixthly, classes should have both private and public storage as well as private and public methods. Public storage can be accessed by subordinate classes, whereas private storage is protected. Public methods are accessible only to the class in which they are encapsulated.

Finally, COBOL classes should be adaptable to compile time parameters which allow attributes and methods to be altered or blended out. This makes it possible to have different variants or mutations of the same class, ensuring reusability.

In summary, object-oriented COBOL programs are

- modularized abstract data types

with informational cohesion and coupling via inheritance and message passing which can be

- adapted at compile time and

- statically or dynamically bound.

(See Figure 1)

4. The REDO Approach to Converting COBOL into an Object-Oriented Specification

Within the scope of a European Research Project on Reengineering named REDO, Oxford University has developed a process for transforming COBOL programs into object-oriented specifications. The inputs to the process are standard COBOL batch programs without database accesses or special data communication interfaces. The result of the process is a formal specification of the programs in the language.
Between the input and output, an intermediate meta-language - UNIFORM - is used as an internal transformation syntax. From this intermediate language, data flow diagrams, entity/relationship diagrams and other technical documents are produced.[6]

The REDO process involves three transformation levels. (See Figure 2)

Level 1 is the translation or re-translation of COBOL syntax back into the intermediate language where it can be more easily processed. Implicit data declarations such as REDEFINES are converted to invariant assertions, and procedural statements are put into a form which includes the data types, e.g.

MOVE X TO Y becomes \( Y := [\text{XXX} \rightarrow 999] (X) \)

where \( X \) has PICTURES XXX and \( Y \) has pictures 999.

This transformation moves the COBOL semantics into a general purpose wide spectrum language system (WSL) which can be used for a number of purposes such as documentation, reengineering, and reverse engineering.[7]

Level 2 is where the object orientation comes in. Every record type is identified as an object and every field as an object attribute. Thus, the Data Division is partitioned into object classes. At the same time, the Procedure Division is cut up into slices based on data flow analysis. Sequences of I/O operations on a particular file, and the intermediate statements effecting the contents of that file make up a program phase. Phases correspond to data flow paths. Since statements may be traversed by many paths they are replicated, so that each phase is a complete sequence of statements from file input to file output.

Level 3 is the generation of an object-oriented specification from the restructured intermediate representation. The program slices are attached to the objects they refer to, becoming methods in a class. All statements which access the record embedded in the class and all statements which alter or set attributes of that record are part of the methods generated. In a final step, the UNIFORM syntax is converted to a Z++ notation.

At the end of the process, there is a class specification for each file and the original procedurally structured statements are now distributed among the classes where they are attached to the object of processing.[8]

5. The REORG Approach to Object-Oriented Reengineering

The REORG approach has grown out of a reengineering experiment at the Union Bank of Switzerland. There, reengineering techniques have already been successfully employed to migrate third-generation applications into the fourth-generation.[9] Now, work is being done to prepare the next transition from fourth-generation to object-oriented.

In the REORG reengineering process, there are 10 steps to transforming procedurally structured COBOL programs into object-oriented ones. (See Figure 3)

1) Static Analysis of source code
2) Inverse Transformation into a specification repository
3) Identification of the object types
4) Analysis of access paths
5) Analysis of data usage
6) Reallocation of instructions to objects
7) Analysis of the data flow
8) Marking of inherited attributes

...
9) Identification of the messages
10) Generation of an object-oriented program frame.

In the first step, the source code is analyzed, parsed, and cut up into diverse symbol tables - for data fields, code blocks, constants, and predicates. In addition, connectivity tables are created for data references, control flow paths, program interfaces, and data interfaces.

In the second step, the contents of the program description tables are loaded via a batch interface into a program specification repository. The COBOL data records are converted to data trees and data dictionary entries. Value assignments are converted to invariant assertions. User and system interfaces are converted into message formats. Database and file accesses are converted into program/object relationships. Subprogram calls are converted into program/program relationships. The control flow structure is converted into a Jackson type tree. The basic code blocks are converted to elementary operations in a function dictionary. The data references are converted to function/data relationships of the type SET, USE, and QUERY.

In the third step, object types are derived from the data structures. Local data structures in the Working Storage Section become objects of type WORK. Data record structures in a File Section become objects of type FILE. Database views in the Working Storage Section become objects of type VIEW. Map and report structure become objects of type INTERFACE-LINKAGE storage structures become objects of type PARAMETER. The result of this step is an object catalogue with pointers from each object to its attributes.

In the fourth step, the file and database accesses are analyzed. The purpose of this analysis is to collect the access operations - such as SELECT, UPDATE, DELETE, INSERT, READ, and WRITE - on each object as well as to recognize the relationships between objects via access sequence. When, for instance, an access to an owner object is followed by an iterative access to a member object, a 1:N relationship can be assumed. When one access to an object is paired with one access to another object, then a 1:1 relationship can be assumed. The access operations are classified as events and are attached to the objects which they access. The relationships are tied both to base and the target object. Upon completion of this step, each object has, besides its attribute tree, a relationship table and a list of access events together with their trigger.

In the fifth step, the data usage is analyzed. For each variable in the DATA DIVISION all the references to that variable are collected from the data reference table and attached to the data item entry in the data dictionary. Thus, each data item description is completed by a list of usage references indicating the program, procedure, elementary operation, and type of usage, whether SET, USE, ALTER, or QUERY.

In the sixth step, the procedural instructions are coupled to the data elements they set or alter. If one instruction affects several variables, e.g.

\texttt{MOVE ZERO TO X,Y,Z}

it is duplicated and attached to each data item. Group operations are assigned to the data group name. Furthermore, all predicates, i.e. IF, WHEN, UNTIL, WHERE conditions, leading to a particular action are included in a path expression attached to the action in question as shown below.
PAYMENT:
UNTIL (I> N) &
IF (OFFICE = FRANKFURT) &
WHEN (LANGUAGE = GERMAN)
ADD VAT TO PAYMENT;

In this example, payment is the targeted data item. Naturally, any one data item can be set or altered many times, thus, having many path expressions of rules.

In the seventh step, the data flow between objects is traced. The purpose of the data flow tracing is to identify those other objects from which any one object derives data, either directly or indirectly. If the other object is semantically superordinate or subordinate, the data flow is considered vertical. For instance, the object DISPATCH-ORDER is subordinate to the objects ORDER and CUSTOMER. It inherits data from both. This is a clear case of multiple inheritance. However, if the other object is semantically at the same level, the data flow is considered horizontal. For instance, the object ARTICLE receives data from the object ORDER, namely the article number and the ordered quantity. This is a clear case of collaboration. As a result of this step, each declared objects has a table of INHERITS FROM and COLLABORATES WITH relationships to other objects. Inherit relationships are depicted in COBOL by PERFORM, collaboration relationships by CALL USING.

In the eighth step, all attributes are marked which are required to be inherited, i.e. those fields in a superordinate object from which values are derived by a subordinate object. For instance, in the statement

MOVE CUSTOMER-ADDRESS TO
DISPATCH-CUSTOMER ADDRESS

the sending field CUSTOMER-ADDRESS belongs to the object CUSTOMER, whereas the receiving field DISPATCH-CUSTOMER-ADDRESS belongs to the object DISPATCH-ORDER. Following this step, there are pointers from all inherited data items to the subordinate classes which require them.

In the ninth step, all attributes are marked which are passed from one object to another at the same semantic level. The sum of such attributes required by an object from another related object make up a message. The point here is to construct import/export messages - export for the sending object and import for the receiving object. A particular message contains all the attributes used or queried by object A, but which belong to object B. For instance, the object ARTICLE requires ARTICLE-NO and QUANTITY-ORDERED from the object ORDER. The result of this step is a table of messages, or parameter lists, for each object/object interface.

The tenth and final step, a new source program in object-oriented COBOL is generated from the intermediate design language. The program structure is a hierarchy of classes, one class for each object extracted from the original program. Subordinate classes have explicit references to data in superordinate classes which they inherit. Data which are inherited are declared in the PUBLIC- STORAGE SECTION of the method which uses or transmits the data. Data used only by that particular class are declared in the PRIVATE- STORAGE SECTION.

The PROCEDURE DIVISION is divided up into a series of METHOD sections. There is a METHOD section for each access to the encapsulated object, i.e. CREATE, DELETE, SELECT, UPDATE, STORE, etc. There is also a section for each attribute or group of attributes altered or set by an event. The section contains all the instructions which alter or set the data together
with their path expressions. The method selection is controlled via event indicators set at runtime. The method parameters are received via the import messages. Generic methods are put into INCLUDE members which can be parameterized at compile-time. The final result is an object-oriented COBOL program according to the latest CODASYL draft. (See Figure 4)

6. Unresolved Problems

A central point of the object-oriented paradigm is the development of nonredundant classes, where a class encapsulates all the attributes and operations of a particular object. In the sample, only one program has been demonstrated. In a large system it would be necessary to repeat the transformation process for every program in the system. Since the same data object may appear in many programs, this means, many variants of the same class will be created. When all of the programs of a system have been transformed, it will be necessary to merge the different variants of each class into a single occurrence. Attributes with the same names or with synonym names, and methods with the same command sequences must be recognized and not repeated. Unfortunately, it is not always easy to recognize similarity, especially in the case of methods. This can lead to the case where a class has different methods producing the same result. There are approaches to solving this redundancy problem, for instance "pattern matching" and "result analysis"[11], but no algorithmic approach is complete or totally reliable. Thus, human interaction is required to remove redundancy and to optimize the final class composition.

Another problem arises out of the collaboration of different classes at the same hierarchical level. The message transmission operations cannot always be properly positioned within a class. Their messages themselves may also be incomplete. Here again, it would be the task of the human reengineer to finish the job. Perhaps, the law of Demeter could help to minimize the collaboration problem.[12]

REFERENCES

at the Union Bank of Switzerland", in Proc. of 7th International Conference on Software Maintenance, Sorrento, Italy, Oct. 1991


Figure 1: Migration from Centralized to Distributed Systems
REDON - PROCESS

1. LEVEL

COBOL - PROGRAM

TRANSFORMATION IN UNIFORM

PROZEDURAL - ORIENTED UNIFORM WIDE SPECTRUM LANGUAGE

2. LEVEL

ANALYSIS & REORGANISATION

OBJECT - ORIENTED UNIFORM WIDE SPECTRUM LANGUAGE

3. LEVEL

TRANSFORMATION IN Z ++

OBJECT - ORIENTED Z ++ SPECIFICATION

Figure 2: REDO Reengineering
**Figure 3**

```
1. Static Analysis
2. Inverse Transformation
3. Procedural Repository
   - Object Identification
   - Access Path Analysis
4. Data Usage Analysis
5. Relocation of Operations
6. Dataflow Analysis
7. Data Delegation Analysis
8. Data Exchange Analysis
9. Inherited Attributes
10. Messages

Figure 3
```
Figure 4: Automated Reengineering Steps