Cobol to Object Oriented Cobol Migration as a Cobol Legacy System Re-engineering Approach

Giancarlo Succhi, Andrea Fossetti and Carl Ulrik

LII/DISA, Università di Trento
Via Zan 8, I-38058 Rovereto, Italia
Phone: +39-464-443140
Fax: +39-464-443141

DIST, Università di Genova
Via all'Opera Pia 11a, I-16145 Genova Italia
Phone: +39-010-353551
Fax: +39-010-353550

Abstract: Cobol is the most used programming language, but often Cobol programs have poor structure and design. This means the maintenance process is a hard-yet-indispensable process, so tools are needed to make it easier and more efficient. This paper explores an option to support efforts in this direction, based on re-engineering and object-oriented technologies, which consists of the migration of Cobol to Object-Oriented Cobol. Because Object-Oriented Cobol will be a part of the imminent new ANSI Cobol standard, the proposed idea represents one of the new possibilities to explore which have been opened by it.

1. Introduction

Before the end of 1997, all being well, the new Cobol (COmmon Business Oriented Language) language ANSI (American National Standards Institute) standard definition will be concluded and the new standard will substitute the previous one of 1985. It is interesting to underline that in the last thirty years, this is the fourth updating of the ANSI Cobol standard from the previous standards published in 1968, in 1974 and in 1985.

The reason of this continued evolution process can be seen in the enormous success that Cobol has registered since its birth and that nowadays still appears to be unassailable. Worldwide it is estimated that there are probably more than two million active Cobol programmers (Resman 1993) and about 80 billion lines of Cobol source code are running to support business, industry, and government. Furthermore consider that, according with these assessments, of the 80 billion lines of existing Cobol code, approximately 60 billion lines (McClure 1992) are unstructured.

Such statistics are relatively indicative of the following two unequivocal and indisputable facts:

- Cobol is the most used programming language around the world.
- A large part of Cobol code used worldwide is not structured.

The main cause for the continued evolution effort to develop new standards for this language is largely connected to the widespread unstructured use of Cobol. There is an impelling need to provide Cobol programmers with a programming platform becoming increasingly modern with features to allow taking advantage of well-acclaimed qualities of Cobol (at least according to the statistics), without risking to produce unstructured, unreliable code containing old fashioned features in comparison with the modern software engineering techniques of the times. Yet, even more important is another problem which dramatically needs to be solved nowadays, the key question of maintenance.

Business applications for which the Cobol language was developed have an intrinsically variable structure, so they need continuous maintenance work having fundamental importance. One can consider, for example, a payroll management system of a common firm around the world: every time the collective labor agreement is revised, the need to correct the payroll system will arise, changing parameters and updating the business rules contained in the system. Maintenance is very expensive work both in money and in time. It is estimated (Tomic 1994) that 80 billion is spent annually on software maintenance and 80 to 80 percent of the working hours of an estimated one million programmers. Software maintenance expenses (McClure 1992) lie in a range from 50 to 90 percent of total software life cycle expenses.

Those who maintain a system often are not the same people who developed it, and this fact causes further problems. It provokes a chain reaction: subsequent maintenance tasks on the same code face not only the very bad quality of the original system but also problems resulting from previous maintenance work performed in an unstructured and uncontrolled way with very serious risks of introducing unpredictable and undesired secondary effects. Needless to say, an unstructured, undocumented, contorted source code makes it very hard to do maintenance. The final results are systems with very short life expectancy that will become unserviceable, or not very reliable in any case, in a few years from their development.
2. Object Oriented Cobol

Against this background, the new ANSI Cobol '97 will take on a great importance, in particular with respect to a very interesting extension it includes: Object Oriented Cobol. At the moment, at the American National Standard Institute, there is one specific committee, the X3J4, working on the preparation of the new standard, and within this committee has been formed a subcommittee, the X3J4.1, which is the Object Oriented COBOL Task Group whose aim is to develop the shape of the object oriented extension. This extension will provide a set of capabilities which will allow the Cobol community to participate in the object-oriented game. The precise specification of Object Oriented Cobol is currently in flux, but it is already possible to understand generally what it will look like.

However, before looking into the basic concepts included in Object Oriented Cobol, it is interesting to note that, among the numerous vendors who demonstrated their interest in this new extension, one can also find big firms like IBM confirming the validity of this idea.

Object Oriented Cobol (Grehan 1994) represents an attempt to reply to what the Cobol community wants. It wants some reassurance that there will be extensions to Cobol that will allow it to take full advantage of the benefits of object-oriented design and development and, simultaneously, it wants abundant reassurance that existing, legacy code will not get broken in the move.

The new main capabilities provided by the extension are the following:

1. The ability to define classes, comprising class object definitions and object definitions.
2. The ability to define data encapsulated inside class objects and objects.
3. The ability to define methods for class objects and objects.
4. The ability to use inheritance and define subclasses.
5. The ability to use polymorphism and interfaces for maximum flexibility.
6. The ability to define data items able to hold references to objects.
7. The ability to invoke methods on objects.
8. The ability to create and manage objects as required.
9. The ability to use objects as a normal part of Cobol programming both in developing new programs and in maintaining existing programs.

It is important to note that in this specification we can find all the main parts composing the Object Oriented Paradigm (OOP) as shown in Figure 1, so one can consider Object Oriented Cobol as a regular object-oriented programming language. Thus, it is reasonable to expect all the advantages of these kinds of languages in using it.

The ability to develop new applications using the object-oriented extensions gives the Cobol community a lot of important new prospects for the future life of the Cobol language in particular considering that the business world complexity is ever growing, and consequently there is an ever growing need to manage information faster and faster in an efficient way.

However in reading the ninth capability listed above, one also has to consider the possible effects for existing legacy Cobol systems. There must be reliable techniques available to migrate them from the old versions or dialects to the new one.

3. Migration as a Reengineering Tool

Facing the dangerous situation in software maintenance created in last years, technology to support this work has continued to evolve, developing a particular branch of software engineering that is called software reengineering. Here software reengineering means the examination and alteration of a subject system to reconstitute it in a new form and the subsequent implementation of the new form (Chikoisky and Cross 1990). An essential part of this technology is automation, the use of automatic tools accessing techniques also borrowed from Computer Aided Software Engineering (CASE) environments. Reengineering is software maintenance automation.

Software reengineering (McClure 1992) targets five main operations on the code:

- Improve its future maintainability.
- Upgrade its technology.
- Extend its life expectancy.
- Capture its components in a repository where CASE tools can be used to support it.
- Increase maintenance productivity.

When a software system reaches such dramatic situations as those mentioned in the introduction, inhibiting its owners' business growth and capacity to change, there are several approaches to consider in trying to solve the problem:

- Do nothing: just continue the status quo.
- Add more people to the system maintenance staff to reduce the software backlog.
- Focus on new development technologies to build new systems that are easier to maintain and, gradually or as soon as possible depending on the strategy preferred, replace existing, aging systems.
- Adopt a direct attack on software maintenance by using automated tools to improve the maintainability and technology of existing systems and the efficiency of the maintenance process: it means reengineering the system.

Looking at this list, one finds only two truly reasonable solutions to the problem: developing a new substitutive system or reengineering the old one. Sometimes the situation is so contorted as to make reengineering infeasible, but usually it is possible to choose. Thus there are two important facts one must consider to drive this choice. The first fact is that the cost of a manual rewrite is $10 to $20 per line of code while the cost of reengineering lies in the range of $0.02 to $2. The second fact, descending from the first one, is that the estimated replacement cost for all the 80 billion lines of Cobol source estimated existing around the world approaches $2 trillion, and this is not a pleasant prospect for the innumerable institutions which based their business applications on Cobol and are faced with the maintenance problem. Thus, reengineering critical systems is often a viable alternative to replacement because it is cheaper, easier and safer.

At this point enters the particular topic of interest which is migration. The current situation in software maintenance has been discussed above and attention has been focused on Cobol and its new capabilities provided by the new standard. Presently it is suggested that the maintenance and object-oriented aspects can be effectively blended with an emphasis on migration.

Migration is a type of reengineering that will play an important role in improving software in the next years. Here migration means the process of converting a software system (program) from one language to another, moving it from one operating environment to another or upgrading its technology (McCusker 1992). Here the focus will be on the language conversion. Such an operation is useful when the new language offers more facilities to the system maintainers in one or more aspects of software maintenance: corrective maintenance, adaptive maintenance or perfective maintenance.

4. Migrating from Cobol to Object Oriented Cobol

As seen, Object Oriented Cobol is a great opportunity not only for new system's development but also for upgrading existing legacy systems, having precision and effectiveness in the migration process while preserving the original system's integrity.

New software technologies are in a continuous evolution process, and entrepreneurial sense suggests that an enormous benefit would be derived from adapting the existing old programs to the new technologies (Greban 1994). Migrating legacy Cobol systems to the Object Oriented extension could satisfy such ambitions.

In general, this kind of work has the ability to produce all the reengineering benefits listed in the previous section if concluded successfully. In fact, a source code written with care in an object oriented language certainly guarantees an increased maintainability and an enhancement in the productivity for the coding part of the software life cycle, in particular considering the enormous reuse potential introduced by object orientation. Similarly, the simple conversion to Object Oriented Cobol offers a real upgrading in the technologies on which applications are based, and therefore extends their life expectancy.

Another fundamental aspect in software reengineering, and in software management in general, is understanding how people and culture affect projects, studying the psychology and sociology of the process (Sharon 1991). Accepting new working methods is a very difficult challenge for programmers or maintainers, and often system managers face this problem in introducing innovations into their own environments. An extension like object orientation for Cobol, not a full conversion, will help: it is certainly easier to introduce something new than radically changing all the working environment. In this way, often people who are used to the old environment existing for many years will accept the new environment sooner and with few difficulties. To the manager, this is important because it means a cheaper, faster and more efficient implementation of the innovation process.

Finally one must keep in mind that migration, like restructuring in general, is not sufficient to perform a complete maintenance operation, but often it is necessary work viewed as preventive maintenance to give code a form which facilitates easier and more efficient global maintenance capabilities. Migration can only improve maintainability; and it does so without changing any of the functionality of code.

5. Developing an automated tool

A model of generic reengineering or reverse engineering tools architecture (Chikofsky and Cross 1990) follows (see Figure 2). This model has general value and it can
be applied to the specific application under discussion.

Initially it is noted that there is no real conversion between any programming languages for the source code of any legacy systems under being considered because the language is not being radically changed, but only being transformed to make use of extensions to the same language. This means that theoretically the program could be left unchanged or, in a more intelligent way, only a part of the source code being worked on should be modified.

The first step in developing a tool to perform the conversion is suggested by the scheme in Figure 2: a parser analyzes the original source code extracting the information needed to recognize parts on which the conversion will focus. This is a key development stage because this operation includes formulating rules to identify which parts of the source code could be expressed better using the object oriented paradigm. In this stage, the evaluation of metric parameters applied to the code is useful too. In fact, it provides the user with additional information about the code to help in evaluating the opportunities for reengineering and the choice of what type of maintenance is needed. When the parser has completed its work, a programmer then has to rewrite some new code modifying the identified parts.

Therefore the problem to solve is twofold. One must define which parts in the old code will be updated and how these parts will be transformed. A similar study has been already reported in two articles ((Markosian et al. 1994), (Tomich 1994)) but these studies only focused on the possibility to modularize Cobol programs in an object oriented manner without considering the migration to an object oriented language. Nonetheless, these results can serve to help us in the conversion.

The tool can build objects forming them from the data elements of the original code and extracting related methods from the procedural code working on these data

![Figure 2: Generic Tool Architecture](image)

![Figure 3: Migration Process Scheme](image)

elements. These new objects will be constructed in the following way:

- Data declarations are properly moved from the old code position to the new object declarations preserving internal structure and alignment.
- Object method invocations are placed in the old code to substitute procedural parts used to build the methods.
- Migration steps are documented to facilitate future maintenance.

When data is identified to build objects, one also has to define these objects following a set of rules:

- Defining objects involves the related class and class object definitions. If a set of objects with similar structure can be extracted from the old code, a single class definition for all of them can be built.
- Defining methods involves interface definitions. Each object has an interface comprising name and parameter specifications for each method supported by the object.
Performing a reverse engineering operation is possible to define encapsulation for data and methods within each object, specifying it in the object definition.

Defining objects and classes is necessary to include local variable declarations in the object or class object Data Division moving them from the old code. It is also necessary to define an object or class object Environment Division.

It is also necessary to declare object identifiers to use objects.

All parts of old code which have not been involved in the object oriented process remain the same; one has only to perform changes to ensure consistency, such as substitute procedural parts used for methods with method invocations.

This whole structure described above is summarized graphically in Figure 3. Various information is needed to perform the migration process. To obtain this information documents found in the source system documentation should be used together with some non-trivial techniques to retrieve additional information (as usually documentation does not cover all topics needed for maintenance or its had quality does not allow effective browsing).

Included in these techniques to extract information are Set-Use Analysis and Program Slicing. Set-Use Analysis provides a global vision of the system to indicate where and how each data element's contents are defined or used. This analysis is necessary to point out parameters for methods or to decide which data elements could be encapsulated in objects. Program slicing could also be useful as the technique allow to isolate program parts containing statements which may not be physically adjacent but are semantically related. Information retrieved in this way are useful in defining each method code.

One can also use results from modularization studies such as the following, to assist in consistently introducing encapsulation:

- If a data element included in an object has a superior item with an OCCURS clause, the superior item is also included.
- If an included data element has an OCCURS clause (including one added by the previous rule), its immediate superior is included.
- If an included data element has a REDEFINES clause, the item it redefines is also included.
- If a conditional name is included, its conditional data element is also included.
- If a procedural paragraph is included in a method, all paragraphs in the PERFORM call graphs below must be selected.

Note that this process is not simple. Determining method parameters, for example, is a hard task because it involves data flow analysis which is complicated by the existence of aliases.

6. Conclusions

The Cobol maintenance problem is a pressing problem. For most institutions using Cobol in their business application, maintenance is not an option but an urgent, sometimes dramatic, need.

This paper has explored a new way proffered to solve the maintenance question, but it comes from work yet in progress. Later it should be possible to yield more refined studies allowing a heavier use of polymorphism and inheritance capabilities included in Object Oriented Cobol to even greater advantage than the simple techniques suggested herein.

References


ANSI X3J4. Object Model Feature Matrix.