A CRITICISM OF COBOL FOR DOCUMENTATION

by
Martin Hopkins

Marty Hopkins is Corporate Technical Director, responsible for Processing Languages. He joined CUC in 1959 and now works in the Mount Kisco office. He has been in the computing field since 1957, when he received his B.A. in Philosophy from Amherst College.

Marty and his wife Camille live in Chappaqua, N.Y. with their two children, Ellen and Andrew.

As COBOL processors appear that compile rapidly and produce fair object code, there has been a resurgence of interest in that language. Therefore, I think it is an appropriate time to raise a word of caution about the virtues of COBOL as a vehicle for documentation. I feel caution is necessary because there are many who claim that the COBOL program itself is all the documentation required.

First, it is possible to write a well documented or poorly documented program in any language. If you don't think this is possible consider a COBOL program written with a maze of ALTERS and PERFORMS and no use of NOTE, and compare it with an intelligently commented assembly language listing. Of course that is an unfair comparison, but it serves to illustrate the point.

But my concern here is: How good is the language for documentation? And does it encourage good documentation practices? In this instance I think, COBOL can be contrasted with an assembly language listing, which is often considered to be a bad vehicle for information about a program.

There used to be a belief that COBOL programs were written by vice presidents and stock boys. This is now generally recognized to be nonsense. Programs are written by programmers no matter what the language. However, there is still a belief that if stock boys don't have the patience or skill to read a COBOL program, perhaps vice presidents and even accountants will be willing to try. However, in any well run shop even this isn't necessary All they have to do is read the write-up which the project supervisor requires.

I feel it is important that people on this level know what programmers are doing. To do this rapidly, a proper write-up of the program is required. Most supervisors simply don't have the time to walk the data through the program. So, the non-programmers are saved – they can read the external documentation. Of course programmers doing maintenance will want more, but their problems will be covered later.

Many people seem surprised at the idea that a well documented COBOL program is really not adequate documentation. Therefore, let me give an example. Supposing a manager asks, "What does this program do?" The answer he gets is a COBOL program that begins:

```
NOTE. CLEAR ALL TABLES TO ZERO.

HOUSE-KEEPING.

PERFORM CLEAR – TAB VARYING I FROM
1 BY 1 UNTIL I EQUALS 100.

PERFORM .....
```

Obviously, a write-up is required. A program (an algorithm) and a description are different things. A program, whether in COBOL or any other language is designed to solve a problem. A description is organized to explain a problem. They just don't look the same. A program is filled with trivia that are not separated from what's important in a description. Part of the problem is: What's an unimportant detail in a description, is a bug in a program. When you consider that most programs reflect their checkout history and changes in design during initial implementation and maintenance, it's hard to think of a worse way to describe a problem.

Of course some technicians must look at programs. For the most part, such listing readers are programmers doing maintenance. But, there will be a few people who will require more than the concise, written description and flow charts. All these poor souls must look at a source language listing for ultimate resolution of details.

If you should come to a COBOL source program after working with reasonable assembly language listings, several immediate flaws in the "higher level" language will be obvious. First, there is no way to skip to a new page in COBOL. This serves to separate portions of a listing and helps mak for clarity. Also, there is no way to skip severa lines with one statement. Another good format feature in many assemblers is the ability to vary the heading and subheading throughout the listing. Assembly language programmers are used to looking for the important comments that are clearly designated with an asterisk. The COBOL NOTE just doesn't stand out.

Finally, there is always the horrible chance that the COBOL programmer took the format rules of the language seriously and wrote this program in free form to make it look like English. After looking at such a program you are thankful that the assembler requires the operation code to begin in column 10, or whatever.

Any attempts to dig into a COBOL listing will reveal further problems in understanding. Many of these arise from the separation of information about a process. Ostensibly the Procedure Division describes what is happening. But in reality many operations are implied in Data Division data descriptions. Conversions, scaling, truncation, removal of signs, editing and variety of padding options are all implied by Data Division clauses. This forces the reader of the Procedure Division to constantly cross reference the Data Division to understand what's going on. This constant flipping back and forth is not the only adverse effect of this remote implication. Since the performance the operation depends on how the data is described, programmers have to redefine areas and assign new names and data descriptions to vary the processing on a particular data field that logically has a single name.

COBOL is not the only problem oriented language with this characteristic. But, with other languages you don't have this problem in assembly language. Even a novice coder can see that the absolute value of a variable is being taken in a program written in assembly language. A COBOL programmer can easily forget that a series of moves to differently named fields actually refers to the same area with the S missing from the picture in one data description.

In addition to the problems of clarity from Data Division implications there is the problem of excessive names. There are just too many to remember. Every programmer who comes from assembly language to COBOL is shocked when he learns how much effort and how many names are required to describe data. Pity the poor reader who is trying to figure out what's going on as the data is manipulated in redefined areas. Then too, the lack of relative addressing causes more names. If a programmer wants to work on the last character of a six character field called BOX, he must redefine the area rather than refer to BOX + 5, ich is beautifully clear The COBOL programmers may call it "BOX-PLUS-5", but this is ambiguous. It just happens that this is a common case where assembly language is much easier to comprehend and, to code.

Another consequence of remote implication is that no person can be sure of the meaning of a MOVE CORRESPONDING without a detailed examination of the data. The consequences of segmentation rules on the ALTER and PERFORM verbs can be a source of misunderstanding. Some switches are reset each time a segment is read in. Some remain set and you can't tell by looking at either the switch or setting statement.

Another problem is: The COBOL language isn't concerned with subroutines. Relocatable loaders, linkage editors, binders (or whatever program that fills in external references, incorporates library subroutines and assigns absolute locations to separately compiled subroutines is called), have become the dominant mode of operation. There are some advantages to this. For instance you can work with smaller modules and therefore, isolate problems. COBOL programs are often too large and diffuse for easy understanding, yet the accustomed method of coping with this problem is not available. (IBM has implemented, through ENTER, something that makes a COBOL program have like a FORTRAN Subprogram. This is wise, not COBOL. IBOL?)

The final objection to COBOL for documentation is psychological. Even if the supervisor insists on a description and adequate use of NOTE, sheer weariness overtakes the programmer who has to code a Data Division larger than a Procedure Division which, in turn, has such abominations as:

> "PERFORM P-20 VARYING I FROM 1 BY 1 UNTIL I EQUALS 10." Rather than "D0201 1, 10."

Can you blame the supervisor if out of charity, he permits a little skimping on documentation? Especially when some "experts" say the program itself is easy to understand.

In order to write well documented COBOL programs, installations should develop standard practices. The following is a suggested list:

1. Make all names unique. This eliminates the CORRESPONDING option but increases clarity and makes maintenance easier. ?
2. Always use the simplest forms. It helps to begin each statement on a new card. ✓
3. Never use compound conditionals or implied subjects.
4. Standardize on data names. It is helpful to indicate a field to be edited by beginning it with an E.
5. Follow each NOTE with an asterisk. In this way they will stand out.
6. Begin each section with a NOTE describing the function of the section.
7. Make all PERFORM loops nest so that there is no overlap. Always use an EXIT.

A COBOL installation will want to extend this list. It will probably develop standard data descriptions with systematically derived names. The guiding principal in all such efforts will be simplicity and the use of an installation prescribed method when there are several COBOL options which will work.

Given a good compiler, COBOL has great value in many situations. However, the COBOL listing is not acceptable as the only documentation because the COBOL language tends to obscure meaning rather than clarify it. Rather than adding such features as an RPG or ADD CORRESPONDING THE CODASYL, a committee should concentrate on adding useful documentation facilities. Meanwhile, the COBOL programmer should provide adequate documentation beyond his COBOL source program.

COMPUTER USAGE COMPANY, INC. 344 Main Street, Mount Kisco, N. Y. 10549