Software Engineering for the Cobol Environment

Michael Evans
Skandinaviska Enskilda Banken

SUMMARY: In an attempt to improve the productivity of their 70 development staff, Skandinaviska Enskilda Banken has built an integrated set of manual and automatic tools for the implementation of Cobol programs. It was possible to use a number of modern programming techniques, including software engineering methods, in a Cobol environment. The project required 31 person-months; the aims, current status, and initial results are reported.

1. Introduction

Skandinaviska Enskilda Banken (S-E-Banken) began programming in Cobol relatively late. By learning from the experiences of other Cobol installations, a number of modern methods were incorporated into the programming environment. This environment has been in use for 16 months.

2. Background

In 1969 S-E-Banken decided to develop online banking applications using assembler language in order to use the hardware then available efficiently. The 1970 vintage hardware was used until 1980.

While application development languages, report generators, and similar products are said to give great improvements in programmer productivity [18], these tools cannot be used for all programs. Therefore, a "safety net" language was needed to take care of the exceptions. In September 1980, the bank decided to use Cobol as its basic programming language.

2.1 Project Limitations

The Jackson Structured Programming [14] technique for program design was already being used. Therefore the "Cobol Programming Environment" (CPE) project only affected the coding and later system development phases.

Limited personnel resources were available for the project. (See "CPE Project Costs," 5.1). This precluded the development of all tools from scratch; instead, software was bought where possible. Project effort was concentrated on integrating the various products plus producing tools which were not available elsewhere.

2.2 CPE Project Goals

The main goals of the Cobol Programming Environment project were to:

(1) assist programmers in producing programs which are more reliable and easier to maintain;
(2) provide a language which is application rather than machine oriented;
(3) take advantage of the methods developed in software engineering during the past 10 years;
(4) produce an educational package which could be used as required;
(5) make the tools for Cobol program development available at the touch of a key;
(6) provide good self-help facilities.

Business programmers often ignore methods which have been available for many years, complaining that academic results are too theoretical and cannot affect their daily work. We wanted to use software engineering methods in a Cobol environment.

As the conversion of program development from assembler to Cobol will take a number of years, programmers must be able to learn Cobol when they are to use it. This led to the need for a self-study package combined with course documentation.

The burden on programmers can be eased if various standards are checked automatically or if certain documentation is generated in a skeleton form. In addition, a note from a tool saying that a particular sequence of statements is inefficient

CR Categories and Subject Descriptors: D.2.6 [Software Engineering]: Programming Environments
Additional Key Words and Phrases: Cobol tools, business programming, commercial programming
Author's present address: M. Evans, Skandinaviska Enskilda Banken, AC/System, Marknad, Datateknik, HK, S-106 40 Stockholm, Sweden.

overcomes the psychological barrier which clever assembler programmers feel when confronted with the new Cobol language.

The assembler programming environment in use had been built up during the past 10 years. If a conversion to Cobol was to be successful, the Cobol environment had to be at least as easy to use as was the assembler one.

3. Manual Techniques

Programming involves manual activities which are also part of the programming environment. The CPE project addressed some of these, with the programming language itself an important factor.

3.1 Cobol Weaknesses

Fosdick [7] identified a number of weaknesses with the Cobol language. The most important of these:

(1) Lack of block structure makes structured programming difficult. If pure Cobol is used for structured programming some code which ought to be inline is forced out of line.
(2) Cobol is verbose.
(3) Local data items cannot be defined.
(4) The syntax for internal and external procedure calls is different.
(5) Cobol does not use functions.
(6) Most mathematical functions are not available.
(7) Various systems programming functions such as bit manipulation, subtasking, and exception handling, are not present.

3.1.1 *Macro Cobol*

We use assembler macros to package a series of machine instructions; Cobol statements need to be packaged in the same way. The MetaCobol macro processor [2] was chosen to support this facility. The Cobol code is maintained at the high macro level and translated to pure Cobol at compile time using MetaCobol. The generated Cobol code is not used by programmers, but is treated with as much disdain as the generated assembler code which Cobol compilers can be forced to list.

```
    ...
PROCEDURE DIVISION.
A-NONSENSE-MAIN-ROUTINE.
*************************************************************
*                                                           *
*    GETS                                                   *
*        NO SPECIAL PARAMETERS                              *
*    DOES                                                   *
*        READS WHOLE FILE PROCESSING ACCORDING TO           *
*        TRANSACTION CODE          #                        *
*    GIVES                                                  *
*        FILE UPDATED (INDIRECTLY)                          *
*                                                           *
*************************************************************
*
     DO  B-INITIALISATION
*
     LOOP
         DO  RA-READ-INFILE
     WHILE NOT F-EOF-INFILE
         SELECT FIRST ACTION FOR I1-TRANS-CODE
         WHEN K-UPDATE-CODE
             DO  C-UPDATE-RECORD
         WHEN K-ADD-CODE
             DO  D-ADD-RECORD-TO-FILE
         WHEN K-DELETE-CODE
             DO  E-DELETE-RECORD
         WHEN NONE
             DO  XA-ERROR-TRANS-CODE
         ENDSELECT
     ENDLOOP
*
     DO  N-CLOSE-DOWN
*
     GOBACK.
```

Fig. 1. A sample of SEBOL source code (Part 1 of 2).

A MetaCobol macro package [3] removes the first three of the above weaknesses by, among other things, implementing control structures for structured programming. This allows the Cobol programmer to have the benefit of a more modern programming language. See [20] for further suggestions on how Cobol may be improved.

3.1.2 *Cobol Extensions*

Interfaces to such service modules as database handlers can be packaged using macros. This allows the call to be written as a Cobol-like verb with operands which may be checked at compile time, instead of just a call statement with parameters [16]. This use of macros was a natural continuation of our previous assembler macros.

The combination of the above items resulted in the definition within S-E-Banken of SEBOL (Skandinaviska Enskilda Banken Oriented Language) as:

- an unofficial subset of ANSI 74 Cobol
+ the control structures required by JSP
+ verb level interface to IMS (Information Management System), IBM's database handler
+ S-E-Banken's extension verbs which, for example, validate check digits

An example of SEBOL code is shown in Figures 1 and 2.

Each ANSI Cobol verb is categorized as:

Recommended. Verbs within the limited subset which are taught in the course. The majority of administrative data processing problems can be solved using these verbs.

Allowed. Verbs which are acceptable but which may cause maintenance problems.

```
 C-UPDATE-RECORD.
*************************************************************
*                                                           *
*     GETS                                                  *
*        RECORD READ INTO I1-RECORD                         *
*     DOES                                                  *
*        UPDATE RECORD WITH TODAY'S DATE                    *
*        (MONTH AND DAY ONLY)                               *
*     GIVES                                                 *
*        UPDATED RECORD WRITTEN                             *
*                                                           *
*************************************************************
*
      START DATA
          01  LC-LOCAL-DATA.
              05  LC-YYMMDD.
                  10  LC-YY   PIC XX.
                  10  LC-MMDD PIC X(4).
              05  FLAG LC-DATE-REFORMATTED IS FALSE.
      END DATA
*
      IF LC-DATE-REFORMATTED IS FALSE
          MOVE W-TODAYS-DATE TO LC-YYMMDD
          SET-TRUE LC-DATE-REFORMATTED
      ENDIF
      MOVE LC-MMDD TO I1-LATEST-DATE
      DO RC-WRITE-FROM-INPUT
      ...
```

Fig. 2. A sample of SEBOL source code (Part 2 of 2).

Forbidden. Verbs which cause serious problems or which are due to be dropped from the CODASYL standard.

The Cobol programming handbook contains the complete syntax of allowed Cobol verbs, including the structured extensions, in the form of a reference card which can easily be taken out and used by programmers.

3.2 Code Skeletons

In order to simplify the writing of new programs, a series of skeleton modules are available in the main source library. Skeletons are provided for the following categories of programs:

- Batch updating—ordinary files and databases
- TP transaction
- Data area definition COPY member
- Various logical read or write modules
- List program

New skeletons have been added as soon as suitable standard solutions for regularly recurring problems have been found. Instead of each programmer having a favorite, possibly erroneous, module, all use the same skeleton, thus spreading standard solutions throughout the company. Each skeleton contains a number of text strings which are changed as the new module is created. These strings are documented at the beginning of each module, as in Figure 3.

3.3 Program Inspections

The technique of inspecting code to find errors has been available for some time. Although we cannot see measurable results, we think that applying the technique to our new Cobol projects will have three advantages:

(1) Fewer errors in delivered programs [8].

(2) Experience of Cobol programming spread quickly throughout the development department.

(3) Feedback to the Cobol support group. By collecting statistics on the number and type of errors found as suggested in [6], parts of the environment may be improved.

3.4 Education of Present Programmers

A four-week course has been written to "convert" assembler programmers to Cobol. The documentation may be used for self-study. It is recommended, however, that all students attend the classroom sessions as soon as possible in order to clear up misunderstandings.

The main emphasis in the course is on the practical application of Cobol. About half the time is used to produce a Cobol program, initially a simple "desk calculator" program. New features are added to it as Cobol verbs are introduced in the course. Writing and modifying this program helps programmers feel happy with Cobol and with using the development techniques.

4. Automatic Tools

Programmers must use certain automatic tools, e.g. compilers. Other tools may be available but their use is optional and often depends on whether they are easily invoked. Ease of use is one of the CPE project goals.

4.1 Compilation

A normal SEBOL compilation consists of the following steps:

(1) MetaCobol's expansion of the various non-Cobol verbs and control structures to ANSI 74 Cobol. See Figures 4 and 5 for examples of the expanded Cobol code.

(2) A Cobol compiler [11].

(3) An optional optimizer step [5] which also performs a static control flow analysis of the Procedure Division. It flags conditions which in reality are unconditional and code which cannot be executed.

(4) A post processor which merges the high level SEBOL source listing from the input to step 1 and the compiler information from steps 2 or 3 to one useful listing. This processor was developed during the CPE project because the compiler listing represents the program and should therefore be easily readable. Among

other things, the compiler output is altered to refer to the source input line numbers instead of to the generated code. Optionally, the listing may be complemented with a short note showing which Cobol verbs expanded from the SEBOL line.

Although the above sounds complicated, compilation is invoked simply through one display panel. Normally only the module name, and possibly a compiler option, must be input in order to initiate a compile. The run's complexity is hidden from the programmer.

4.2 Standards Checking

A MetaCobol macro package [1], modified to work with S-E-Banken's standards, can check each module for violation of standards. Forbidden verbs are flagged. Warnings are output for functions which are not recommended, for example ACCEPT and DISPLAY. Code which is inefficient is flagged. This provides self-help for experienced as well as inexperienced Cobol programmers.

The prettyprinter, which is another version of the same package, indents code in the Procedure Division to show the control structure. Level numbers in the Data Division are standardized and the definitions of numeric items are tidied up.

```
 IDENTIFICATION DIVISION.
*XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
*                                                                 X
*    L I S T - P R O G R A M                                      X
*                                                                 X
*    VARIABLES FOR "FIND" AND "CHANGE" :                          X
*                                                                 X
*       %MOD            MODULE NAME                               X
*       %IOMI           I/O-MODULE NAME INPUT FILE                X
*       %IOML           I/O-MODULE NAME LIST FILE                 X
*                                                                 X
*       %PROG           PROGRAMMER NAME                           X
*       %DAT            YEAR AND MONTH FOR CODING                 X
*                       FORMAT YYYY-MM                            X
*                                                                 X
*       %?              EVERYWHERE WHERE COMMENTS MUST            X
*                       WRITTEN OR NAMES OR PREFIXES              X
*                       SUPPLIED                                  X
*                                                                 X
*    REMOVE THIS BOX WHEN ALL CHANGES HAVE BEEN MADE              X
*                                                                 X
*XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
 PROGRAM-ID.      %MOD.
*************************************************MCSLISTP*
*                                                                 *
*    MODULE       %MOD                                          *
*                 %? TEXT DESCRIPTION                           *
*                                                                 *
*    FUNCTION     %?                                            *
*                                                                 *
*    CODED        %DAT                                          *
*                                                                 *
*    CODED BY     %PROG                                         *
*                                                                 *
*****************************************************************
/
 ENVIRONMENT DIVISION.
 CONFIGURATION SECTION.
 SOURCE-COMPUTER. IBM-370.
 OBJECT-COMPUTER. IBM-370.
 SPECIAL-NAMES. DECIMAL-POINT IS COMMA.
/
 DATA DIVISION.
```

Fig. 3. Introduction to the list program skeleton.

4.3 Interactive Debug

An interactive debug product [12] is available. It contains the following commands:

- List fields
- Change (most) fields
- Set conditional or unconditional breakpoints
- Trace execution
- List source statements

The module being tested must first be compiled with a special option. The compiler outputs symbol tables and a copy of the module source code, which are then input to the debug monitor. The details of which compiler tables the monitor requires are hidden from the programmer behind a single panel invocation, greatly simplifying the use of this tool.

4.4 Database Simulator

The execution time interface to IMS, IBM's database product, is complex. This interface may be simulated [10] during program development, allowing transactions to be built up, database calls to be traced, and program execution monitored. Although this product may be used for all programming languages, it is required for the development of database programs and must therefore be integrated into the Cobol programming environment

4.5 Execution Profile

Optimizer III [5] collects execution profile information as counts and/or timings. A number of runs for one module may be combined to report a total execution profile. This allows the programmer to see the degree of test coverage, although the coverage measure, e.g., C0 or C1, is not reported explicitly.

In addition, one report lists all statements which are executed the same number of times, for example, per transaction or master record. The program's logical structure can be seen from this information.

4.6 Update Log

A menu option is provided to compare old and new versions of the same module, listing the total

changes made between them. A modified version of the Compare program [19] is used. This ignores line sequence numbers and only compares the Cobol A and B margins character by character. At present, Cobol syntax information is not used, so that lines containing additional blank characters are regarded as changed although they are syntactically identical.

4.7 Data Area Listing

The listing of common, COPYed, data areas is suppressed during compilation. Only the area's text description and version number are put into the compiler listing. The Cobol definition and compiler data map for an area may be listed by a separate run which is only needed once per program instead of the information appearing in every compiler listing.

4.8 Procedure Hierarchy

The MetaCobol structured programming package uses internal Cobol procedures—paragraphs which are PERFORMed. A documentor cross-references PERFORMS to procedures and then lists the complete PERFORM tree. We have added another report showing each procedure with its PERFORMS. Figures 6 and 7 include examples of these reports which provide an automatic equivalent to hand-drawn tree diagrams. The mechanical listings may be produced as required and are always up to date

4.9 Implementation

All tools are available through SPF [13], the menu oriented display monitor which is used for normal program editing. Panels, menus, and the links between them may be customized.

All tools are listed according to the results produced, not the program products involved. The programmer does not need to know which specific programs are used to produce each result. This means we

```
WORKING-STORAGE SECTION.
    ...
01  LC-LOCAL-DATA.
    05 LC-YYMMDD.
        10 LC-YY        PIC XX.
        10 LC-MMDD      PIC X(4).
    05 ZSPP-LC-DATE-REFORMATTED PICTURE X VALUE 'F'.
        88 LC-DATE-REFORMATTED VALUE 'T'.
PROCEDURE DIVISION.
A-NONSENSE-MAIN-ROUTINE.
    ...
    PERFORM B-INITIALISATION THRU B-INITIALISATION-EXIT.
A-NONSENSE-MAIN-ROUT-LP-1.
    PERFORM RA-READ-INFILE THRU RA-READ-INFILE-EXIT
    IF NOT F-EOF-INFILE
    NEXT SENTENCE ELSE
    GO TO A-NONSENSE-MAIN-ROUT-LX-1.
    IF I1-TRANS-CODE = K-UPDATE-CODE
    NEXT SENTENCE ELSE
    GO TO A-NONSENSE-MAIN-ROUT-SC-2-101.
    PERFORM C-UPDATE-RECORD THRU C-UPDATE-RECORD-EXIT
    GO TO A-NONSENSE-MAIN-ROUT-SS-2.
A-NONSENSE-MAIN-ROUT-SC-2-101.
    IF I1-TRANS-CODE = K-ADD-CODE
    NEXT SENTENCE ELSE
    GO TO A-NONSENSE-MAIN-ROUT-SC-2-102.
    PERFORM D-ADD-RECORD-TO-FILE THRU
    D-ADD-RECORD-TO-FILE-EXIT
    GO TO A-NONSENSE-MAIN-ROUT-SS-2.
A-NONSENSE-MAIN-ROUT-SC-2-102.
    IF I1-TRANS-CODE = K-DELETE-CODE
    NEXT SENTENCE ELSE
    GO TO A-NONSENSE-MAIN-ROUT-SC-2-103.
    PERFORM E-DELETE-RECORD THRU E-DELETE-RECORD-EXIT
    GO TO A-NONSENSE-MAIN-ROUT-SS-2.
A-NONSENSE-MAIN-ROUT-SC-2-103.
    PERFORM XA-ERROR-TRANS-CODE THRU
    XA-ERROR-TRANS-CODE-EXIT.
A-NONSENSE-MAIN-ROUT-SS-2.
    GO TO A-NONSENSE-MAIN-ROUT-LP-1.
A-NONSENSE-MAIN-ROUT-LX-1.
*
    PERFORM N-CLOSE-DOWN THRU N-CLOSE-DOWN-EXIT.
A-NONSENSE-MAIN-ROUTINE-EXIT.
    GOBACK.
```

Fig. 4. Cobol code generated by the above SEBOL code (Part 1 of 2).

```
/
C-UPDATE-RECORD.
    ...
    IF LC-DATE-REFORMATTED
    GO TO C-UPDATE-RECORD-IF-1-100.
    MOVE W-TODAYS-DATE TO LC-YYMMDD
    MOVE 'T' TO ZSPP-LC-DATE-REFORMATTED.
C-UPDATE-RECORD-IF-1-100.
    MOVE LC-MMDD TO I1-LATEST-DATE
*
    PERFORM RC-WRITE-FROM-INPUT THRU
    RC-WRITE-FROM-INPUT-EXIT.
C-UPDATE-RECORD-EXIT.
...
```

Fig. 5. Cobol code generated by the above SEBOL code (Part 2 of 2).

Table I. Functions available under the COBOL menu.

- ◦ Coding
  - Prettyprinting
  - Compilation
  - Standards checker
  - Link edit
- ◦ Test
  - Interactive debug
  - Execution profile analysis
- ◦ Documentation
  - Procedure hierarchy within module
  - Source code comparison
  - Listing of COPYed data areas
- ◦ Miscellaneous
  - Generate SCHEMA

Table II. Resources required to develop the major tools and aids.

Tool	person-months
Handbook	4.0
Course	5.3
Skeleton modules	0.6
Menus, panels	2.5
SEBOL macros (MetaCOBOL)	7.5
Listing postprocessor	2.0
Standards checker and pretty-printer	2.0

can add an extra listing to the procedure hierarchy run without having to tell everyone how to produce the listing.

All available functions are presented on a basic Cobol menu (Table I). This reminds the programmer which facilities are available. The number of parameters needed to run each function is minimized because SPF remembers often-used values and sets intelligent defaults. An example of the parameters needed to perform a standards check is in Figure 8. Behind each menu option is a series of tutorial screens, telling the programmer what the function does and which parameters are required.

5. Experience Thus Far

This programming environment has been used regularly since April 1981. The development costs and experience gained while using it are reported below.

```
NONSENSE   P R O C E D U R E   H I E R A R C H Y

    01  A-NONSENSE-MAIN-ROUTINE
      02  B-INITIALISATION
      02  RA-READ-INFILE
      02  C-UPDATE-RECORD
        03  RC-WRITE-FROM-INPUT
      02  D-ADD-RECORD-TO-FILE
      02  E-DELETE-RECORD
      02  XA-ERROR-TRANS-CODE
      02  N-CLOSE-DOWN
```

Fig. 6. Full procedure hierarchy listing: Note that C-UPDATE-RECORD is expanded to include the PERFORM of RC-WRITE-FROM-IN-PUT.

```
NONSENSE   C A L L   S T R U C T U R E
    DEFINITION                REFERENCE

    A-NONSENSE-MAIN-ROUTINE
                              B-INITIALISATION
                              C-UPDATE-RECORD
                              D-ADD-RECORD-TO-FILE
                              E-DELETE-RECORD
                              N-CLOSE-DOWN
                              RA-READ-INFILE
                              XA-ERROR-TRANS-CODE
     C-UPDATE-RECORD
                              RC-WRITE-FROM-INPUT
     B-INITIALISATION
     D-ADD-RECORD-TO-FILE
     E-DELETE-RECORD
     N-CLOSE-DOWN
     RA-READ-INFILE
     RC-WRITE-FROM-INPUT
     XA-ERROR-TRANS-CODE
```

Fig. 7. PERFORMs per procedure.

```
---------------  SEBOL Standard Checker -----------------

Source Library:                 Schema Library (for IMS):
  Project ===> SE                 Project ===> SE
  Library ===> TEST               Library ===> COB
  Type    ===> SOURCE             Type    ===> SCHEMA
  Member  ===>                    IMS-module ? => Y (Y/N)

LIST id ===>                TERM listing desired ===> N (Y/N)
SYSOUT class ===> *

          PROBLEM?  press the 'HELP' key
```

Fig. 8. Panel for standards check: As shown to the programmer. Only member name must be supplied in order run the function. All other values have been previously supplied by this programmer, but may be altered if desired. The new values are then saved.

5.1 CPE Project Costs

The project lasted six months. Eight people were involved, spending a total of 31 person-months working on it. (Table II shows activities on which the most time was spent.) Seven of the people had a programming background, although only four of them had previously actually programmed in Cobol. Program products were bought and rented from various suppliers for about US $60,000.

The continuing implementation of Cobol at S-E-Banken requires one person full-time. Most of this time is

used for holding courses and consulting about application projects.

5.2 Education

The first two courses were held with classroom lessons in a traditional manner. Later courses have used more time for self-study with only five hours spent in the classroom.

Thus far, 37 programmers have completed the Cobol course and are able to produce working programs. The educational package fulfills the project goal of teaching Cobol to assembler programmers.

5.3 Programming Environment Statistics

Statistics are kept on how often some of the automatic tools are used. The results of the first 16 months' production programming in Cobol are shown in Table III.

Both the prettyprinter and the standards checker contain severe limitations which prevent their being used on modules which access databases. This is reflected in their limited use thus far. However, programmers feel that similar tools which work correctly would be useful (Table IV).

The procedure hierarchy is used as modules reach production status, to obtain part of the final documentation. It is not normally used during development.

We have not been able to obtain statistics about the manual parts of the environment; for example, which are the most common errors found during code inspections. A survey was conducted among those who have completed the Cobol course (Table V). Most programmers who have previously used assembler feel that the programming environment for Cobol is more helpful.

Table III. Use of the automatic tools: For all production projects during April 1981 to August 1982. Compilation errors are only those detected during macro expansion. Errors detected by the COBOL compiler are not reported.

Tool	Modules	Runs	% Runs in error
Compiler	213	6,712	18
Prettyprinter	26	136	52
Standards check	28	56	5
Procedure hierarchy	41	164	52
Data area listing	20	233	13

Table IV. Usability of the various COBOL tools: Result of a survey among S-E-Banken's COBOL programmers.

	Used in % of modules	Perceived errors	Potential usage % of modules
Course documentation	73	some	77
COBOL handbook	69	some	85
Code inspections	27	some	94
Compiler listing	95	few	94
Prettyprinter	31	many	45*
Standards checker	19	many	59*
Dynamic flow analysis	14	some	45
Interactive test	28	many	67
Database simulator	39	few	55
Procedure call hierarchy	52	few	70
COPY area listing	8	some	63
Compare old vs new	1	none	64*

* indicates tools where the potential usage is very different for new development and for maintenance. The other tools are equally usable for development and maintenance.

Table V. Usability of the COBOL environment: Results of a survey answered by those with COBOL and assembler experience.

	Assembler environment			COBOL environment	
	much better	better	equal	better	much better
Answers	0	2	0	4	2

Note. The two people who found the assembler environment easier to use commented that this was probably due to their limited experience with COBOL.

5.4 Problems

The main problems encountered thus far can be divided into two groups:

(1) the connection between the generated pure Cobol that is input to the various tools and the high level SEBOL code which programmers maintain; and

(2) integration of the various program products.

5.4.1 *Relationship of SEBOL to Cobol*

As the programmer maintains code at the SEBOL level all references to statements ought to refer to this code. This is possible if the generated Cobol code can be matched with the appropriate SEBOL line. However, certain tools renumber their output without regard for the input sequence numbers, making it very dif-

ficult to refer the reports they produce to the input Cobol module, let alone to the original SEBOL module. Also, COPYing area definitions, either explicitly or automatically through database calls, generates many Cobol lines which have no equivalent in the SEBOL source code. Identifying and removing these lines has caused problems in the compilation post processor and the prettyprinter.

5.4.2 *Integration of Various Products*

Although the various products contain functions which are required, it has proved difficult to provide all functions at all times.

Examples of the problems:

•The interactive test package requires a very special execution environment. For example, functions to obtain a profile of paths executed cannot be used while testing interactively.

•The database simulator and the interactive debugger did not work together despite their coming from the same supplier. Much effort was expended in solving this problem.

•All modules must be recompiled between being tested interactively and being run in production.

•The prettyprinter formats database calls and parameters incorrectly.

6. Future Plans

Three directions for the future development of the Cobol programming environment can be seen. They are:

(1) further integration of the available tools;

(2) the use of a data dictionary which is being introduced into systems development activities and which will also affect programming in Cobol; and

(3) the development of new tools.

6.1 Integration of Tools

We intend to increase the use of the high level SEBOL code where this is economically feasible. This will probably be done with filters [15] which reformat listings in much the same way that the compiler post processor does now.

6.2 Interface with a Data Dictionary

We are collecting information about our data and programs for a data dictionary [17], which is to be an active tool for systems development. This agrees with [9] which points out the need for a "software engineering database." S-E-Banken has decided that our data dictionary is to be that database.

Once information is entered into the dictionary it can be extracted in various useful ways. For Cobol programmers, this means that COPY area descriptions for records, database segments, and internal work areas may be produced automatically. In addition, an interface between MetaCobol and Datamanager [4] allows individual field definitions to be retrieved from the dictionary at compile time. The dictionary can be updated from the module's source code [4] using information about which fields, records, and files are used and how.

6.3 New Tools

Further information is available in the Cobol source code. We have defined three reports which would be useful:

(1) a cross-reference of chosen data areas across a number of modules;

(2) a module hierarchy within a program; and

(3) inconsistent parameters across a CALL statement.

6.3.1 *Cross-reference Between a Number of Modules*

One data area, for example a database segment or a program communication area, is often used in a number of different modules. A combined cross-reference listing of the usage of this area in all modules would be beneficial. This may be obtained directly from the source code or, as suggested in [9], from the data dictionary.

6.3.2 *Module Hierarchy Within the Program*

Information about which modules call other modules is available within the linked executable program. This information can be extracted, reformatted, and used to update the data dictionary, from which it can be reported in the same format as the procedure call hierarchy. The resulting documentation would show the actual program structure, removing the need for hand-drawn tree diagrams.

6.3.3 *Checking of Call Parameters*

Cobol does not check the consistency of parameters passed between modules. Details of the parameters are, however, available in the source code. In the future they will be validated and put into the data dictionary.

7. Summary

With a reasonable amount of effort, a number of useful software engineering tehniques can be implemented in a Cobol programming environment. The main considerations when doing this follow:

•Availability. Much of the information required may already be available in another form. Look for it.

•Ease of use. Programmers must regard the new tools as a positive addition to their working environment.

•Integration of the various program products within a Cobol programming environment. The tools must cooperate, not oppose each other.

•Follow-up on tool usage. Build in a method of logging when and how each tool is executed.

•Which departures from ANSI Cobol are accepted by company policy?

Acknowledgments

The author wishes to acknowledge the efforts of all the other members of the CPE project team in converting many odd ideas into a useful set of tools: Ann-Sofie Dean–the standards checker and prettyprinter; Anna Ekelin–educational material; Gunnar Pira–compiler listing post processor; Thomas Risberg–standards and handbook; Lars-Anders

Rolfhamre–the interactive panels which made the whole system usable; and Marianne Larsson for helping us to get the documentation into human readable form.

References

1. Applied Data Research. COBOL/QDM utility procedures guide. Form no. SM2G-07-00, Applied Data Research, Princeton N.J., 1979. Describes the MetaCobol package for quality control of Cobol programs. This package includes standards checking, flagging of inefficient code and tidying the Data Division.

2. Applied Data Research. MetaCOBOL user guide. Form no. SM2G-00-10, Applied Data Research, Princeton N.J., 1979. A general description of the functions provided by the MetaCobol macro translator.

3. Applied Data Research. Toward COBOL structured programming. Form no. SM1G-51-00, Applied Data Research, Princeton N.J., 1976. A summary of the design principles behind the MetaCobol structured programming package and an overview of the Cobol extensions which it supports.

4. Bjergis, L. *MetaCOBOL Interface to a data dictionary.* Spadab, Stockholm, Sweden, 1982.The description of a two-way interface between MetaCobol and Datamanager, implemented as MetaCobol macros plus an exit routine. The interface may be used to 1) fetch data item definitions from the dictionary at compile time and 2) update the dictionary with information from the Cobol source code.

5. Capex. "Optimizer III OS user guide." Form no. SO3-1279-310. Capex Corp., Phoenix, Ariz. 1980. The product description consists of three parts: 1) compile time actions including static control flow analysis, 2) test monitoring and 3) test run analysis including path coverage.

6. Fagan, M. Design and code inspections to reduce errors in program development. IBM Syst. J. *15*, 3 (1976), 182–211. The code inspection technique as applied to application systems development is described. Specific check lists for inspections and examples of the use of inspection results to improve future work are included.

7. Fosdick, H. Opting for PL/1. *Computerworld 14*, 32 (Aug. 11, 1980), 27–30. The relative strengths and weaknesses of PL/1 and Cobol as languages for the development of administrative programs are discussed.

8. Glass, R. Real time: The 'lost world' of software debugging and testing. *Comm. ACM 23*, 5 (May 1980), 264–271. A survey of tools and techniques for testing real time programs including conclusions about how effective the tools are for finding errors.

9. Howden, W. Contemporary software development environments. *Comm. ACM 25*, 5 (May 1982), 318–329. Examines the differences between software development environments depending on investment. Functions which should be present in more advanced environments are defined.

10. IBM. Batch terminal simulator II program description, operation manual. Form no. SH20-1844, IBM, Palo Alto, Calif., 1979. A description of the data base simulator used to test Information Management System (IMS) programs.

11. IBM. COBOL compiler and library programmers guide. Form no. SC28-6483, IBM, San Jose, Calif., 1979. A complete description of the optional processing available in IBM's Cobol compiler, together with a guide to the various compiler reports.

12. IBM. COBOL interactive debug terminal user's guide and reference. Form no. SC28-6465, IBM, San Jose, Calif., 1975. Lists the facilities present in the interactive debugging monitor for IBM Cobol and describes how these are used.

13. IBM. System productivity facility dialog management services. Form no. SC34-2036, IBM, San Jose, Calif., 1980. Shows how standard functions for handling CRT panels, menus and tables may be used to provide a customized programming development user interface.

14. Jackson, M. Principles of program design. Academic Press, New York, 1975, Chap. 4. A detailed description of the Jackson Structured Programming technique which uses the input and output data structures to derive the program structure. The technique is widely used in Europe for developing administrative programs.

15. Kernigham, B., and Plauger, P. *Software Tools.* Addison-Wesley, Reading, Mass., 1976, pp. 35–66. Shows how tools for program development may be built by combining a limited number of basic building blocks. One class of such building blocks is the file tailoring filter.

16. Magee, P. Call statements are harmful but can be controlled. *ACM SIGPLAN Notices 16*, 2 (Feb. 1981), 83–88. Explains how interfacing errors in Cobol may be detected earlier in the development cycle by using a preprocessor which recognizes the interface as a Cobol-like verb, instead of using a standard Cobol CALL statement with a number of unverified parameters.

17. Management Systems and Programming. *Datamanager Fact Book.* MSP, London, England, 1979. A general description of the data dictionary functions which are in the Datamanager product.

18. Martin, J. *Application Development without Programmers.* Savant Institute, Carnforth, England, 1980. Presents the results, including productivity figures, of projects which allowed end users to develop their systems using application generators and very high level languages.

19. Miner, J. Software tools: S-1 compare. *Pascal News 13* (June 1978), 20–23. The Pascal source code of a program which lists the differences between two files, taking account of inserted or deleted records.

20. Weinberg, G., Wright, S., Kauffman, R., and Goetz, M. *High Level COBOL Programming.* Winthrop, Cambridge, Mass., 1977. A discussion of the technical and business aspects of Cobol programming, resulting in suggestions to extend the Cobol language in order to ease program development. These extension could easily be implemented using a preprocessor.

Computing Practices readers are directed to this month's Technical Correspondence on "Another Approach to Data Compression" by Martin Gorfinkel which refers to the paper by Michael Pechura in the September issue, and "On Computer System Messages" by H. Karmin referring to the Pracnique by Ben Shneiderman, in the September issue.