A Method for Reducing Software Life Cycle Costs

W. O. Paine and J. C. Holland
Quality Assurance DSN, MCCC and Mechanical Hardware Section

The advent of new hardware and software tools permits a new approach to preparing, presenting, and maintaining software specifications and corresponding source programs. The method described could significantly reduce many of the associated costs compared with techniques now in general use in industry.

I. Introduction

Software life-cycle costs typically include all of the manpower, equipment usage, and documentation costs necessary to design, implement, and maintain computer software throughout its life. Indirect costs, which are often excluded from this grouping, are the net consequences to operations resulting from the use of the "finished" software. Thus, if software is placed into use while still seriously in error, the indirect cost to operations could be greater than the typical total life cycle cost. On the other hand, correct and well designed software, delivered and used in a timely manner, could yield a significant saving to an operational activity. Effective ways of reducing typical costs while maintaining or improving software quality should lead to an improvement in indirect costs.

When software costs are closely examined, it is seen that skilled manpower is now the major contributor and is the most rapidly increasing cost (Ref. 1). Within the range of such human activities, it appears that designing and testing efforts are the major parts and that the underlying planning, analyzing, and communicating aspects of this effort are major portions. Thus, any real reduction in software life cycle costs must make highly skilled human effort more efficient, especially on large, complex, and long-lasting software projects. When these skilled human functions are examined closely, it is seen that critical factors include the way one communicates with oneself and then with others and finally with computers. Thus, there has been a long and active proliferation of programming languages and, more recently, a number of programming design languages to supplement older methods, such as flowcharting. One way to achieve cost reduction is to provide documentation tools whose use increases the effectiveness of skilled humans (Ref. 2).

II. Proposed Method

A. Conceptual Principles

The conceptual principles that must be met are:

(1) The design document and resulting computer code should be as closely connected as possible, both physically and in descriptive terms.

(2) The design should be readily understood by the non-programming "user" or "client" who provided the original requirements.

(3) The design and code should be readily understood and found comparable by all programmers involved in the preparation and verification of the software throughout its life.

(4) The complete design and source program document should be in a machine-readable form, such that it may be readily stored and processed on one computer. At some point it should be kept on a common single storage device such as a tape or a disk. Thus, it should be possible to also move a copy of the entire machine-readable document at once to an alternate processor for more convenient use, special testing or printing.

(5) Document updating methods should force some consideration of possible updating of both the design and source code on every occasion.

(6) There should be an automatic annotation on each page of the document showing the date of the update version number, and clearly identifying changes made from the previous version.

(7) Pagination and other systematic identification to reveal design structure should be at least semi-automatically produced for accuracy, consistency, and cost-saving convenience.

(8) The method should be independent of the choices made for programming languages and target computers.

B. Hardware and Software Tools

The fairly recent advent of important new hardware and software tools permits the solution shown in Subsection C. However, other tools are available which would allow other approaches to satisfying the conceptual principles given in Subsection A. In any event, a few new and/or expanded software tools would be required to achieve the systematic, operational use of the proposed method. The cost of producing the new software tools should be a very small fraction of the potential savings and certainly would be significantly less than the cost of a compiler or an assembler.

In the proposed method, tools which were conveniently at hand were used to illustrate the approach. Perhaps better or more suitable tools may become available later. The key hardware tool employed is the IBM 3800 (laser) printer. It is described in Refs. 3 and 4. However, for our purposes, it is enough to note that it permits 20 type fonts, up to 204 characters per line and up to 12 lines per vertical inch (2.54 cm). The laser (non-impact) principle yields printing of exceptional quality, approaching that of true graphics arts. The number of multiple copies needed may be specified by a single card with each use, and bursting and trimming are also under program control. The overall capability and flexibility of this printer would allow entire finished detailed design documents, including graphics, to be done at one time.

The key software design tool employed for this example is a program design language currently available on the JPL UNIVAC 1108 called SDDL (for Software Design and Documentation Language) (Ref. 5). In an operational system, code positioning would be either semi- or completely automatic as code is developed to fulfill design statements.

C. Details of the Proposed Method

Three new modes of displaying software under development, or as permanent documentation for field-released versions, are shown in Figs. 1 through 3.[1]

The first of the display modes (see Fig. 1) offers the ability to explicate the nature of the program and its structure in terms of natural language. This feature has the great advantage of eliminating the need for condensing identifiers into obscure mnemonics or restricting the commands to a set of reserved words known to any specific compiler or assembly language. The invented delimiters "DO.UNTIL" and "END.DO" of Fig. 1 serve to indicate a loop within the CMSP1 program, a fact established by a simple directive to SDDL (discussed below).

The flexibility of this language is demonstrated by the definition of the word "EXECUTE" as an indication of calling a subroutine, a fact indicated by the horizontal arrows terminating upon empty parentheses. It is assumed here that the documentation and programming are still under development, and the subroutines referred to have not yet been defined. Words such as "SET" in the instruction "SET STOP = FALSE," which might in some cases be viewed as auxiliary and only for the purpose of aiding readability, are admissible in such a flexible language as SDDL, just as is the usage of the "END.IF" to delimit the logic of an "IF ... THEN ... ELSE ... ENDIF" paragraph. It should be noticed also that the auxiliary word "THEN" is omitted in Fig. 1. The invented delimiter "FINIS" marks the termination of each program module.

[1]Figures 1, 2, and 3 are constrained by printing space in this publication. In operational practice the limits would permit up to 204 characters per 13.6-in. (34.5-cm) line and either 6, 8 or 12 lines per inch (2.54 cm).

The SDDL processor automatically indents the text presented to it in a manner which reflects the underlying structure, as set forth by the definitions of modules, blocks, and escapes within that language. The user may override or modify this indentation by the use of further directives, if he so desires.

The second mode of display afforded by the combination of SDDL and the capability of the IBM 3800 printer is that of the combined SDDL and generated object code – in an interleaved manner – as shown in Fig. 2. The SDDL language allows for the separation of source statements as an arbitrary point designed by a pound sign (#). The portion of the statement to the right of the pound sign is right-justified on the printed output, and is reserved for comments (i.e., has no effect on the structuring of the SDDL statements, as seen by this language processor, with respect to modules, blocks or other structures). The object-language statements, in whatever language is appropriate, may be coordinated with the SDDL statements by placing them in these fields. Auxiliary directives to the SDDL processor may be used to justify certain fields of the object language to certain margins for the sake of uniformity.

The third mode of presentation of the documentation, as shown in Fig. 3, involves a side-by-side listing of the program design language and the completed object code. Here the SDDL statements may, if desired, contain code numbers allowing coordination of these statements with the finished object code. Because of the presence of embedded code modules which may be developed separately (and displayed separately, as in the second mode described above), the final object program and its associated SDDL code will not necessarily present the simple correspondence afforded by the interleaved listing of the previous mode. The use of identifying numbers would, nevertheless, allow the two forms of the same program to be associated in a statement-by-statement manner.

III. Advantages of the Method

The method, when compared with many existing techniques, offers several advantages. By placing both the structured design and resulting code on a single page such that they may be read separately *or* together meets the needs of all users of the document while reducing production and maintenance costs. The designer and programmer, while working in both the development and maintenance modes, are encouraged to read, compare, and maintain both parts of the finished document. It is expected that the clarity and convenience alone would make the method attractive to both end users and implementers. The auditing of code vs design (Ref. 6) would become more efficient. Finally, a step is taken in the direction of allowing future semi-automation in software validation and verification, by virtue of the fact that such a well-structured presentation lends itself well to the processing of inductive assertions.

References

1. Gilb, T., *Software Metrics,* Winthrop, Cambridge, Mass., 1977.
2. Tausworthe, R., *Standardized Development of Computer Software*, SP 43-29, pp. 1-8, Jet Propulsion Laboratory, Pasadena, Calif., July 1976.
3. *Introducing the IBM 3800 Printing Subsystem Audit Programming*, IBM Manual GC26-3829-4, November 1976.
4. *IBM 3800 Printing Subsystem Programmer's Guide*, IBM Manual GC26-3846-1, September 1976.
5. Kleine, H., *SDDL-Software Design & Documentation Language*, Publication 77-24, Jet Propulsion Laboratory, Pasadena, California, May 15, 1977.
6. Holland, J. C., and Paine, W. O., "An Error-Minimizing Software Audit Technique," in *The Deep Space Network Progress Report 42-32*, pp. 201-221, Jet Propulsion Laboratory, Pasadena, California, Apr. 15, 1976.

```
                                                          SSD-DMC-5084-0P
                                                          MODULE NAME     CMSP1
                                                          DATE          7/15/76
                                                          REV     A     9/15/76
PROGRAM  CMSP1

 EXECUTE INITIALIZATION                   ----------------------------------->(( ))

 DO.UNTIL ENDRUN = TRUE

   EXECUTE FIRST.PROGRAM.STAGE            ----------------------------------->(( ))

   IF STOP = TRUE

     SET STOP = FALSE

   ELSE

     IF COMMAND.VALIDATION = TRUE

       EXECUTE SECOND.PROGRAM.STAGE       ----------------------------------->(( ))

     ELSE
     END.IF

     IF STOP = TRUE

       SET  STOP = FALSE

     ELSE

       EXECUTE THIRD.PROGRAM.STAGE        ----------------------------------->(( ))

     END.IF
   END.IF
 END.DO

 EXECUTE TERMINATION.PROGRAM              ----------------------------------->(( ))

FINIS
```

TITLE	DATE	INT'S
PREPARED		
CHECKED		
APPROVED		

Fig. 1. Program design language sample

```
                                                           SSD-DMC-5084-0P
                                                           MODULE NAME      CMSP1
                                                           DATE           7/15/76
PROGRAM  CMSP1                                             REV     A      9/15/76

 EXECUTE INITIALIZATION                          PREEX  EQU $        INSERT CODE

 DO.UNTIL ENDRUN = TRUE                          CMS20  EQU $

   EXECUTE FIRST.PROGRAM.STAGE                   PS1    EQU $        INSERT CODE

   IF STOP = TRUE                                       TBMB,FLAG    STOP,CMS60

     SET STOP = FALSE                                   OBMM,FLAG    STOP

   ELSE

     IF COMMAND.VALIDATION = TRUE                CMS60  TBMB,FLAG    VALDTE,CMS80

       EXECUTE SECOND.PROGRAM.STAGE              PS2    EQU $        INSERT CODE

     ELSE
     END.IF

     IF STOP = TRUE                              CMS80  TBMB,FLAG    STOP,CMS120

       SET  STOP = FALSE                                OBMM,FLAG    STOP
                                                        BRU          CMS130
     ELSE

                                                 CMS120 EQU $
       EXECUTE THIRD.PROGRAM.STAGE               PS3    EQU $        INSERT CODE

     END.IF
   END.IF
 END.DO                                          CMS130 TBMB,FLAG    ENDRUN,CMS20

 EXECUTE TERMINATION.PROGRAM                     CLOSE  EQU $        INSERT CODE

FINIS
```

TITLE	DATE	INT'S
PREPARED		
CHECKED		
APPROVED		

Fig. 2. Program design language with interleaved code

```
                                                              SSD-DMC-5084-0P
                                                              MODULE NAME    CMSP1
                                                              DATE          7/15/76
                                                              REV    A      9/15/76
PROGRAM  FIRST.PROGRAM.STAGE

    SET INITIAL.STAGE = NOT COMPLETED                      OBMM,FLAG    PRERTD

    ESTABLISH INTERRUPT.SUPPORT.PROCESSOR.1.TASK
                                                           REX,EST
                                                           DFC          0
                                                           DFC          @IP1
                                                           DFC          80
                                                           DFC          @LM
                                                           STM,2        IP1RCB

    ESTABLISH BACKGROUND.REAL.TIME.1.TASK
                                                           REX,EST
                                                           DFC          0
                                                           DFC          @BR1
                                                           DFC          90
                                                           DFC          @LM
                                                           STM,2        BR1RCB

    ACTIVATE INTERRUPT.PROCESSOR.1.TASK
                                                           REX,ACT
                                                           DFC          0
                                                           DFC          @IP1
                                                           DFC          0,0

    DO.UNTIL INITIAL.STAGE = COMPLETED              PS130  EQU $
                                                           TBMB,FLAG    PRERTD,PS150
                                                           HOP,PS180

        RELINQUISH                                  PS150  REX,RLNQ

    END.DO                                                 BRU          PS130

    DE-ESTABLISH  BACKGROUND.REAL.TIME.1.TASK
                                                    PS180  REX,DEEST
                                                           DFC          0
                                                           DFC          @BR1
                                                           ZRR,1
                                                           STM,1        BR1RCB

    DE-ESTABLISH  INTERRUPT.PROCESSOR.1.TASK
                                                           REX,DEEST
                                                           DFC          0
                                                           DFC          @IP1
                                                           ZRR,1
                                                           STM,1        IP1RCB

 FINIS
```

TITLE	DATE	INT'S
PREPARED		
CHECKED		
APPROVED		

Fig. 3. Parallel program design language and code