Partially Automated In-Line Documentation (PAID): Design and Implementation of a Software Maintenance Tool

Jane E. Huffman

Science Applications International Corporation

Clifford G. Burgess

University of Southern Mississippi

ABSTRACT

Large scale budget and schedule overruns of software projects have sparked interest in software maintenance. The importance of accurate technical documentation to the maintenance process is presented. A possible solution to the lack of adequate technical documentation is suggested -- a partially automated in-line documentation system. This system uses software metrics to determine where comments should be placed in source programs. The system can be used as a tool in software development and/or software maintenance.

KEY WORDS: Software Engineering Software Maintenance Software Tools Software Metrics Documentation In-Line Documentation

1. INTRODUCTION

Large budget and schedule overruns on many software projects has generated a great deal of interest in software maintenance. Maintenance accounts for approximately 80% of the overall life cycle cost of a software product [10]. Also, personnel dissatisfaction with maintenance work has reached epidemic proportions. The national average length of employment for a computer programmer is approximately six months. Turnover in personnel results in high training costs and added maintenance costs (due to the large number of programmers working on a system over its life cycle). If these costs can be minimized, the overall cost of software can be reduced substantially.

A major factor affecting software maintenance is documentation. Software documentation is the collection of documents that explain, describe, and define the purposes and uses of a particular software program or a system composed of multiple programs [8]. There are three categories of software documentation: user documentation, product documentation, and technical documentation. Technical documentation will be the focus of this paper.

Technical documentation refers to information on the various phases of the software life cycle. It includes design specifications, performance specifications, functional specifications, development information, etc. This documentation is usually written by the system designers or programmers. It is found in-line in the source program, on-line, or in hard copy form. It is often used as reference material for maintenance of a program/system.

Unfortunately, quite often no technical documentation is produced. In addition, when documentation is produced, it is often poorly or incompletely written, and may not be kept current. These factors contribute to the difficulty of maintaining the software at a later time.

2. RESEARCH RATIONALE

This paper will show that undocumented (or poorly documented) source code can be documented with the help of a partially automated in-line documentation system, PAID. This system seeks to improve software maintenance by facilitating the in-line documentation of programs. The system uses software metrics to help determine where comments should be placed in the program. Specifically, the Index of Difficulty used by the Maintainability Analysis Tool (MAT) [1],[2] has been modified for use by PAID. An extended discussion of PAID follows.

3. AUTOMATION OF IN-LINE DOCUMENTATION

Purpose

The importance of technical documentation to the software maintenance process has been discussed, and the need for commenting existing programs is apparent. The ideal solution to this problem would be some system that automatically places comments in a program explaining its operation. This would allow maintenance programmers to submit all of the old programs that need modification (but lack documentation) to this system and within a short period of time have fully documented programs.

Method

PAID concentrates on the major functionality of this "super" documentation system - deciding where comments should be placed. This becomes difficult since no national or international standards for documentation exist. If standards did exist, however, the system could just follow their precepts and guidelines, and place comments as appropriate.

Since no standards exist, PAID implements the following in-line documentation guidelines. In general, comments should be inserted:

1) at the beginning of the program to give the name of the author, title of the program, object of the program, and methods used by the program;

2) in the declaration section to explain variables, data types, record structures, etc. used in the program;

3) at the beginning and end of each block; and

4) within the program and/or block body as deemed necessary (particularly in sections of code that are notoriously difficult to maintain, such as recursive routines).

There are more specific documentation "guidelines" that vary depending on the language used to implement a program.

The first three guidelines can be fairly easily implemented by lexically scanning the subject program and checking for the program heading, declaration section, or beginning/end of a block. Guideline four presents more of a problem -- determining that a segment of code needs documentation requires close examination of the code. PAID uses the concept of textual complexity to make this determination. By implementing a measure of textual complexity, PAID chooses locations in the program where comments should be inserted.

Software metrics are tools used to quantify software complexity. When deciding where to place in-line comments, textual complexity is of importance. Textual complexity deals with the readability and understandability of programs. There are two major metric systems used to quantify textual complexity: 1) the Berry-Meekings style metric; and 2) the Maintainability Analysis Tools' (MAT) [1], [2] Index of Difficulty. The latter will be of interest in this paper.

MAT, developed by Science Applications International Corporation, is a static analysis tool for FORTRAN programs. It is used to locate programming discrepancies such as poor usages, errors, possible errors, etc. This FORTRAN static analyzer reads, parses, and examines the source code of each program module one at a time - i.e., the static analyzer examines each source module individually and all of them as a whole [2]. Weights and factors assigned to program elements and attributes (Index of Difficulty) are used to examine the modules.

PAID utilizes a modified version of the Index of Difficulty, referred to as the Modified Index of Difficulty, to decide where comments should be placed. The Index of Difficulty was selected as the foundation for PAID because it is a measure of how difficult a program will be to maintain and it uses element weights and factors to determine this complexity measure. This makes the metric relatively easy to implement, easy to understand, and easy to modify.

Using the Index of Difficulty weights as a foundation, weights for Pascal source code were determined. This process involved using the authors' experience, peer input, and trial and error. A subset of the weighting factors for the Modified Index of Difficulty are shown in table 1.

PAID examines a line of the source code (subject code) being evaluated. The line is scanned and token types are determined. These token types each have a weight in the Modified Index of Difficulty table. A simple table "look up" procedure is used to retrieve the token metric value. The weight for the token type is then added to the complexity of the line.

If the line's complexity is sufficient to warrant documentation (it exceeds a limit for complexity per line of code), PAID "looks ahead" and "looks behind" an appropriate number of lines of code (based on complexity) to see if such documentation already exists. If it does not, the user is prompted to enter a comment. If the line complexity does not warrant documentation, it is added to the accumulated complexity (the complexity since an already existing comment was found or a new comment was inserted). If accumulated complexity exceeds a certain limit, documentation is necessary, and PAID prompts the user to enter a comment.

PAID checks for complex Pascal structures, also. An example is the forward declaration. This structure makes a program difficult to understand, and therefore difficult to maintain (particularly if the structure is not documented). If a forward declaration is encountered by PAID, it "looks ahead" and "looks behind" for documentation. If documentation does not exist, PAID prompts the user to enter a comment. When prompting for a comment, PAID informs the user of the reason that documentation is necessary (e.g., line complexity, accumulated complexity, forward declaration, etc.).

PAID checks for direct and indirect recursion. Direct recursion is present when a block calls itself. An example of direct recursion implemented in Pascal is:

```
Procedure T;
  begin
    T;
  end;
```

This is often represented as: T --> T. Indirect recursion exists if a block calls another block(s), at least one of

Table 1. Modified Index of Difficulty Weights

Identifiers	Weight
Array	1.50
File Variable	1.00
Function	2.00
Parameter	1.00
Procedure	2.00
Program	2.00
String	2.00
Type	7.00
Variable	0.85
Constants	
Boolean	0.00
Character	0.10
Integer	0.00
Real	0.05
Operator Element	
And	0.25
Not	1.00
Or	0.25
Relational	0.20
+	0.20
-	0.20
/	0.30
*	0.25
Data Types	
Array	7.00
Boolean	7.00
Char	7.00
Integer	7.00
Pointer	7.00
Real	7.00
Record	7.00
Text	7.00
Statement Type	
Assign	2.00
Assignment	0.00
Begin	0.00
Case	5.00
Comment	0.00
Const	0.00
Else	0.00
End	7.00
For	7.00
Forward	7.00
Function	7.00
Go to	10.00
If	5.00
Procedure	4.00
Program	5.00
Readln	3.00
Repeat	1.00
Type	0.00
Until	7.00
Var	0.00
While	7.00
Writeln	3.00

which calls the original block. For example:

```
Procedure Z; forward;
Procedure L;
 begin
  Z;
 end;
Procedure B;
 begin
  L;
 end;
Procedure Z;
 begin
  B;
 end;
```

This can be represented as: Z --> B --> L --> Z.

PAID evaluates the relationships between blocks in the subject program. It compares each identifier it encounters to the current block name. If these match, direct recursion exists and the user is prompted to enter a comment (if a comment does not already exist). The identifier is also compared to all blocks currently active in the program that may have an indirectly recursive relationship to the current block. If indirect recursion is found and no previous comment exists, the user is prompted to enter a comment.

The Jackson design methodology was used to develop PAID in a top down manner. This design methodology was selected due to the authors' experience and familiarity with it. It was decided that PAID would evaluate Pascal programs since the authors have more experience with and knowledge of this language than any other, and because it is the universal teaching language. The resulting PAID design was implemented in VAX Pascal on a VAX 11/780 at the Hattiesburg campus of the University of Southern Mississippi.

Validation

PAID was used to document six existing Pascal programs, two of which will be discussed here. Program Ash is a Pascal program that has examples of high statement complexity, direct recursion, and indirect recursion. Program Ash was evaluated by PAID, and as a result had many comments added.

The original file, Ash.Pas, is shown in figure 1. Notice the direct recursion in Procedure Edit and Function Nextone. Also note the indirectly recursive relationship of Procedures Z and L. The almost total lack of in-line documentation in the program is also noteworthy.

The file output from PAID, Ash.PAID, is shown in figure 2. This program has had 24 comments inserted, 14 of which document BEGINs and ENDs. PAID correctly pointed out the two instances of direct recursion and indirect recursion. It also pointed out statements of considerable complexity to a maintainer, such as the two forward declarations of procedures. PAID produced the desired results by prompting the user to enter comments at appropriate locations in the code.

```
PROGRAM ASH(INPUT, OUTPUT);

VAR
  X,Y: REAL;

PROCEDURE ASK;
 BEGIN
  READLN(X);
 END;

FUNCTION NEXTONE(PP: INTEGER): INTEGER;
 BEGIN
  NEXTONE:= NEXTONE(PP + 1);
 END;

 PROCEDURE Z; FORWARD;
PROCEDURE L; FORWARD;

PROCEDURE WANT;

 PROCEDURE EDIT;
  BEGIN
   EDIT;
  END;
 BEGIN
  EDIT;
  Z;
 END;

PROCEDURE Z;
 BEGIN
  WANT;
  L;
 END;

PROCEDURE L;
 BEGIN
  Z;
 END;

BEGIN
  ASK;
  READLN(Y);
  Y := NEXTONE(Y);
  WRITELN(Y);
  Z;
END.
```

Figure 1. Ash.Pas

Comments inserted at PAID's request can be differentiated from previously existing comments in two ways: 1) all PAID comments have { } delimiters; and 2) all PAID comments are lower case only. PAID asks the user for preferences of comment locations before evaluation starts. The user may request comment insertion after every BEGIN and/or END or request that BEGINs and ENDs not be commented at all.

Figures 3 - 5 show PAID's operation. Figure 3 shows PAID eliciting user preferences on comment locations. Figure 4 shows PAID prompting the user to document a section of code containing indirect recursion. Figure 5 shows the ending statistics for the evaluation of Ash.Pas.

```
PROGRAM ASH(INPUT, OUTPUT);

VAR
  X,Y: REAL;

{x and y are occurrence counters}
{ procedure ask will ask user for data}
PROCEDURE ASK;
 BEGIN {ask}
  READLN(X);
 END; {ask}

{function nextone will return the next value}
FUNCTION NEXTONE(PP: INTEGER): INTEGER;
 BEGIN {nextone}
  NEXTONE:= NEXTONE(PP + 1);
{recursive call to nextone}
 END; {nextone}

 PROCEDURE Z; FORWARD;
PROCEDURE L; FORWARD;

{ these procedures z and l will be found later in the
program}

{want will ask user for data}
PROCEDURE WANT;
{edit will call itself}
 PROCEDURE EDIT;
  BEGIN {edit}
   EDIT;
{ recursive call to edit}
  END; {edit}
 BEGIN {want}
  EDIT;
  Z;
{ indirect recursion}
 END; {want}

PROCEDURE Z;
 BEGIN {z}
  WANT;
  L;
{indirect recursion}
 END;{z}

PROCEDURE L;
 BEGIN {l}
  Z;
 END; {l}

BEGIN {main}
  ASK;
  READLN(Y);
  Y := NEXTONE(Y);
  WRITELN(Y);
  Z;
END. {main}
```

Figure 2. Ash.PAID

```
Would you like comments after every BEGIN and END?

comment after (B)EGIN
comment after (E)ND
comment after both of the (A)bove
do not place comment after either of (T)hese

Your choice:
```

Figure 3. Comment Location Preference

```
SOURCE CODE LINES     23 THROUGH  26.
The following code should be commented due to
INDIRECT RECURSION
In particular, line     4 requires documentation.

     1    END;

     2   BEGIN

     3    EDIT;

     4    Z;

Enter Y if you wish to enter a comment or S to see these
lines of code again, otherwise enter any other letter.
y
Enter parenthesized letter for where you want comment
placed or any other to continue.

(H)elp - more information.
(B)efore these lines of code.
(A)fter these lines of code.
(C)omment before line of your choice.
(O)n line   4 after the statement.
```

Figure 4. PAID Comment Prompt due to Indirect Recursion

```
Average line complexity was  2.42.
There were  1 previous comment(s) found.
There were  6 comment(s) inserted.

Source program has    46 lines of code.
Original program had  46.00 lines of code per comment.

Program now has  6.57 lines of code per comment.
```

Figure 5. PAID Ending Statistics

Utilmod.Pas, not shown to conserve space, is a module from a Pascal compiler. It is comprised of several different procedures whose functions range from inserting a symbol into the symbol table to generating assembly code. This program, as compared to Ash.Pas, already contained in-line documentation and was much larger in size (in terms of lines of code) and much broader in scope and nature. Ash.Pas illustrated PAID's ability to document programs that are documentation deficient. Utilmod.Pas demonstrates PAID's ability to document programs that already contain documentation.

Utilmod.Pas, prior to being submitted to PAID, contained two major types of comments: 1) comments before each block; and 2) comments after the END of each block. This documentation may be sufficient for relatively simple blocks. However, for more complex blocks, such as Save_Id and Process_For, additional documentation is warranted. In these instances, PAID is used to "fine tune" and supplement existing comments.

After submission to PAID, the output file (Utilmod.PAID) contained 29 new comments. Of these 29 comments, 10 documented BEGINs and ENDs. PAID had been instructed to document all BEGIN and END statements prior to evaluating the program.

Validation is defined as substantiating, or confirming that the desired result is produced. If standards for in-line documentation existed, PAID could be validated by comparing a program it documented to these standards. Since no such standards exist, PAID must be validated using the "standards" it was developed to implement. By examining the test programs evaluated by PAID (including the programs Ash.Pas and Utilmod.Pas) before and after being submitted to PAID, it can be seen that the four in-line documentation guidelines (presented in section 3) are satisfied.

4. FUTURE USES OF METRICS IN SOFTWARE MAINTENANCE

Software metrics offer hope for simplified software maintenance. A standard for software metrics would be a giant step forward for software maintenance. Many applications of metrics are currently being researched. PAID suggests the application of metrics to the partial automation of in-line documentation. Such a system could be used alone or in conjunction with a program generator.

PAID offers the possibility of several other software engineering applications. The concepts implemented in PAID could be expanded and/or modified to:

1) statically analyze programs and tag or mark locations that need comments. Management personnel could run PAID on pseudocode, evolving code, or code that needs updating and get an estimate of the amount of work required for documentation. This tool could help managers estimate the cost (particularly in manhours) of developing or maintaining software;

2) analyze programs written in different programming languages. By adding knowledge bases (weight table like the Modified Index of Difficulty) and modifying the lexical scanner, other programming languages besides Pascal could be evaluated;

3) include a knowledge acquisition component. PAID could then develop a "model" of each user and would know where the user commonly places comments. Also, the user would be allowed to modify the knowledge base and make it more specific to the current application; and

4) measure the quality of code being developed. This measure could be used to monitor programming teams and to pinpoint weak(/strong) team members. Such a system would help a manager prevent poor code from being incorporated in a developing system, thus improving the system's maintainability.

Metrics could be applied to the complexity analysis of unmodularized code. Unstructured programs could be analyzed and module decisions suggested based on the complexity of sections of code. Maintainability effort can be measured using software metrics with systems similar to MAT. Such information assists managers/programmers/analysts in making programs easier/less costly to maintain.

Metrics have been used largely during the coding phase of the software life cycle. If metrics are applied to earlier phases of the life cycle, several controlling factors can be determined:

* complexity of the software at that phase
* ways to reduce complexity
* successful completion of the phase
* manpower allocation necessary to meet deadlines

Although much more research is needed in the area of software maintenance, the work thus far indicates that rapid advances can be expected. With increased development and use of software engineering tools, such as interactive viewing systems, reusable code, program generators, metric driven systems, and structured design techniques, the software maintenance problem can be contained. Finding ways to decrease the cost of maintenance and the dislike for maintenance work is a vital issue in Software Engineering today and deserves serious attention.

REFERENCES

[1] Berns, Gerald M. (1984). Assessing software maintainability. Communications of the ACM, 27(1), 14-23.

[2] Berns, Gerald M. (1985). Analysis tool tracks down bugs in FORTRAN code. Computer Design, June, 169-174.

[3] Berry, R .E. and Meekings, B.A.E. (1985). A style analysis of C programs. Communications of the ACM, 28(1), 80-88.

[4] Boehm, B.W. (1973). Software and its impact: A quantitative approach. Datamation, April.

[5] Curtis, Bill, Sheppard, Sylvia, Milliman, Phil, Borst, M.A., and Love, Tom (1979). Measuring the psychological complexity of software maintenance tasks with the Halstead and McCabe metrics. IEEE Transactions on Software Engineering, SE-5(2), 96- 104.

[6] Ejigou, Lem O. (1984). A simple measure of software complexity. Computerworld, April 2, 1984, 10/10 - 10/16.

[7] Halstead, M.H. (1977). Elements of software science. New York: Elsevier.

[8] Houghton-Alico, Doann. (1985). Creating computer software user guides: From manuals to menus. New York: McGraw-Hill Book.

[9] McCabe, T.J. (1976). A complexity measure. IEEE Transactions on Software Engineering, SE-2, 308-320.

[10] Wiener, Richard and Sincovec, Richard (1984). Software engineering with Modula-2 and Ada. New York: John Wiley & Sons.