Using Literate Programming to Teach Good Programming Practices

Stephen Shum
Computer Science Department
Augustana College
Sioux Falls, SD 57197
shum@inst.augie.edu

Curtis Cook
Computer Science Department
Oregon State University
Corvallis, Oregon 97331-3202
cook@cs.orst.edu

Abstract

The ability to comprehend a program written by other individuals is becoming increasingly important in software development and maintenance. In an attempt to encourage undergraduate Computer Science students to write informative and usable documentation, the literate programming paradigm was incorporated into the teaching of one undergraduate Computer Science course at Augustana College. This paper describes the concept of literate programming, the experience of using literate programming to teach good programming practices, and the results from the experiment that showed that literate programming encourages more documentation.

Introduction

The ability to comprehend a program written by other individuals is becoming increasingly important in software development and maintenance. Studies have shown that 30-90% of software expenditure is spent on maintaining existing software [15, 12]. Studies have also shown that maintenance programmers spend about half of their time studying the code and related documentation. This has led Standish[12] to conclude that the cost of comprehending a program is the dominant cost of a program over its entire life cycle.

A survey by Chapin[4] of maintainers showed that they perceived poor documentation as the biggest problem in software maintenance work. Poor documentation has one or more of the following characteristics [9, 4, 8] :
a. Nonexistent and incomplete.
b. Inconsistency between code and documentation.
c. Difficulty in finding information.
d. Not appropriate for all levels of programmer experience.

There are a number of CASE tools available that claim to satisfy the documentation needs of software maintenance. These tools generate automatic documentation in the form of reports by static analysis of source code. Examples of documentation produced are: control flow chart, data flow chart, cross-reference listings, metric reports, call graphs, module hierarchy chart, etc. All of this information is helpful to maintenance programmers to become familiar with the structure of a program and to navigate around the program during maintenance investigation. What this documentation fails to provide is insight into why a particular program structure is used, or how the program functions. It also fails to provide information on other relationships between program components other than these syntactical relationships [4].

As computer science teachers we have the responsibility to teach students about the importance of informative and usable documentation as well as readable code. However, students usually see documentation and sometimes the design process as non-essential and extra compared to coding. They do not see the need of documenting a program and writing readable code. One method we are using to encourage more readable programs is to assign a relatively high percentage of the programming assignment grade to documentation and readability. In fact, the grading scheme used by the author at Augustana College in all of his computer science courses: 25% each for correctness, design, readability, and documentation. Such a grading scheme has been found to be quite effective in forcing students into the habit of using meaningful names and effective program formatting, including internal comments, and providing external program design documentation in the form of a pseudocode. However, the quality of the documentation is far from being informative since most of the comments are usually commenting the obvious. In addition, most students admit that they generate the pseudocode document from the program code after the program has been tested.

Literate programming [7] seemed an ideal solution to the problem of teaching good documentation practices and helping students write informative and usable documentation. Literate programming emphasizes writing programs that are intended to be read by humans rather than a computer. A literate program is a single documentation (file) containing both documentation and program. It presents the code and documentation in an interleaved fashion that matches the programmer's mental representation. Program design, design and code decisions, and other useful information for the reader are included in a literate program. One key feature of literate programming is that the source code and documentation are created simultaneously. Hence students writing literate programs see the importance of documentation and clearly see the relation between the design and code.

To test these ideas and the claim that literate programming encourages more and better documentation we introduced literate programming into a junior level computer science programming in Spring semester 1992. The AOPS (Abstraction Oriented Programming System) literate programming system [11] was used. For an experiment in the class students did two program assignments in AOPS and two in Turbo C. This paper describes the experiences teaching literate programming and results from the experiment.

Literate Programming

Knuth [7] coined the term "literate programming" to emphasize writing
code that is intended to be read by humans. He believes that the format and structure of a program should be designed to communicate primarily with the humans who read the program rather than the computers that execute the program. The presentation of the code should proceed according to the mental patterns of the author/programmer rather than the patterns demanded by the language and compiler. He claims that programming in this way produces better programs with better documentation.

The literate programmer writes a source file with the program code and documentation interleaved. The literate paradigm recognizes that two different audiences, human readers and compilers, will receive the program. For the human readers, there is a filter program that transforms the source file program code and documentation into a form that can be processed by a document formatter. For the computer, there is a filter program that produces a source code program suitable for input to a compiler from the source file.

It is important to note that in literate programming program code and documentation are created simultaneously. Literate programming systems contain mechanisms for associating documentation with sections of code and give the programmer freedom to write the program sections in an order that maximizes program comprehension. Hence instead of being appended as extra information to the program, documentation is viewed as an integral part of the program. Its omission, rather than its inclusion, is what is most noticeable.

WEB

To promote the literate programming concept, Knuth [7] developed the WEB system as an example literate programming system. In WEB a programmer writes a source file with the program code and documentation interleaved. WEB has two filter programs: Weave and Tangle. Weave transforms the source file into a form that can be processed by TEX. Tangle extracts the program code from the source file and puts it into a form that can be input to a Pascal compiler. This process is shown in Figure 1.

```
                  Weave                   TEX
                  ---------> file.human -------------> typeset
                  |                      listing
                  |
        file.web ---|
                  |
                  |                     compiled
                  ---------> file.machine -----------> program
                  Tangle                  Pascal
                                              compiler
```

Figure 1. The Processing Paths in Web.

AOPS (Abstraction Oriented Programming System)

Several Web-like literate programming systems have been developed since 1984 [1, 5, 6, 10, 13, 14, 15]. However, all of these systems including WEB are programming language and typesetting language dependent, and they do not allow flexible listing and viewing of program code and documentation. Although AOPS [11] is WEB-like, it is programming language and typesetting language independent. Among the other features of AOPS are a flexible lister which allows users to print selected portions of the program code and/or documentation and a hypertext browser which allows users to display relevant information on the screen while suppressing irrelevant details.

AOPS Program

An AOPS program is written in levels of abstractions. It consists of AOPS rules defining the highest level abstraction and all the abstractions used directly and indirectly by the highest level abstraction. A rule consists of an abstraction name, equal sign, type and a body. The abstraction name (hereafter referred to as AO-name) is a string of characters of any length delimited by a character not used in the AOPS source file for any other purpose. (We will use the at sign, @, in our examples.) There are three basic types of AOPS rules.

1. The Code Rule

The program code portion of an AOPS literate program is constructed from code rules of the form:

AO-name=code AO-body

where AO-body consists of legal statements of the underlying programming language with embedded AO-names. (Think of an AO-name embedded in an AO-body of a code rule as a macro call.) The code rule in essence embodies the goal-plan structure of computer programs. The AO-name specifies the goal and the AO-body specifies the plan used to achieve the goal. AOPS provides the ability to explicitly describe this goal-plan structure in a very natural way that is not restricted by the syntax of the programming language. For example, using

Pascal as the programming language, the code rule for @8-queens program@ is defined as:

```
    @8-queens program@=code
    program eightqueens;
        var @variables of 8-queens@
    @procedures and functions of        8-
queens@
        begin
            @8-queens solution@
    end
```

There are code rules that define each of the three abstractions, @variables of 8-queens@, @procedures and functions of 8-queens@, and @8-queens solution@. These may occur before or after the @8-queens program@ rule.

2. The Textdoc Rule

To help readers understand the code definition of an abstraction, AOPS allows a programmer to associate textual or graphical documentation with the abstraction name. Textual documentation such as design decisions, alternate solutions, or anything that will help readers comprehend the code refinement is defined by a textdoc rule of the form:

AO-name=textdoc AO-body

where AO-body is a string of any characters. The textdoc rule (and the graphicdoc rule described below) is completely invisible to the compiler of the programming language. Hence an AOPS user can use his or her favorite word processor to typeset the documentation. For example, the textdoc rule for @8queensprogram@ is:

```
  @8-queens program@=textdoc     Description:
Given are an 8X8 chessboard   and   8   queens
which    are hostile to each other.  Find a
    position for each queen such that      no
queen may be taken by any other   queen,   i.e.,
every row,    column, and diagonal contains at
    most one queen.
    Input: none
    Output: The positions of the 8    hostile
queens
```

3. The Graphicdoc Rule

One major criticism of Knuth's several literate programs is the lack of diagrams. This is not because one cannot incorporate figures and diagrams in a WEB program, but more likely due to the fact that WEB does not encourage programmers to include pictorial documentation. AOPS, on the other hand, encourages users to include pictures and diagrams by providing a special rule for graphical documentation and allowing users to use their favorite word processors to compose the pictures. The graphicdoc rule has the form:

AO-name=graphicdoc AO-body

where AO-body is a pictorial illustration. Figure 2 gives an example of a graphicdoc rule.

AOPS imposes no restriction on the ordering of the rules so that an AOPS program can be designed and developed in an order or style preferred by the programmer free of the restrictions of the programming

@8-queens program@=graphicdoc

One acceptable solution to the 8-queens problem is:

Q							
						Q	
				Q			
							Q
	Q						
			Q				
					Q		
		Q					

Figure 2: Example of graphicdoc rule.

language syntax. Hence an AOPS programmer may break a task into subtasks, and tackle the subtasks in whatever order he or she prefers.

Experience Teaching Literate Programming

In Spring 1992, we used AOPS in one of our course offerings. The course syllabus was basically not affected by this incorporation except one class period was used to discuss AOPS.

AOPS fit in nicely with teaching program design. AOPS implicitly suggests the design methodology Functional Design (FD) expressed using a pseudocode PDL (Program Design Language). Both FD and PDL are widely practiced and taught in undergraduate programming courses. AOPS treats the PDL design embedded as code rules and supports a smooth transition form design to code. Hence students no longer see PDL design as an extra and nonessential step but as a natural step in the software development process, even for small problems.

The Experiment

The experiment was designed to test one of the claims of the literate programming paradigm: that literate programming encourages more documentation.

Hypothesis: Programmers are likely to include more documentation, as measured by the ratio between the number of comment characters and the number of code characters, when using a literate programming system than when using a traditional programming system.

Independent and Dependent Variables: The independent variable is the programming system used: literate versus traditional. The dependent variable is the ratio (number of comment characters) /(number of code characters), although the ratios (number of comment words)/(number of code words) and (number of comment lines)/(number of code lines) were also computed and analyzed.

The reason that these ratios were used instead of simply the comment counts is that the amount of comments should be in proportion to amount of code. That is, it seems reasonable for programs with more code to contain more comments. Comment counts alone ignore the amount of code. So even if literate programs are found to have more comments than traditional programs, it could be because they have more code than traditional programs and not because programmers are likely to include more comments using a literate system. The statistic (comment character)/(code character) gives the number of comment characters associated with each code character.

Design: In order to control the wide individual variability, a within-subjects design was used.

Each subject saw both experimental conditions which served as a control for his or her own performance.

Subjects: Our subjects were 16 students enrolled in the class. Ten subjects were seniors, three juniors, and three post college. As a part of the class, they were required to learn and write programs using AOPS and to write programs using Turbo C.

Procedure: The subjects were randomly divided into two groups, A and B, by matching the students according to their GPA. The two students in each of the matching pairs are randomly assigned, one to group A and one to group B. As part of the class all students were instructed in the use of AOPS. Two programming assignments were given. Group A was to do the first assignment in AOPS and the second in Turbo C while Group B was to do the first in Turbo C and the second in AOPS. Both groups were given the same problem specifications. Both assignments were to follow the same project guidelines handed out at the beginning of the semester. They had ten days to do each assignment. Graded assignments were not handed back until after both assignments were turned in.

During that three-week period when the students are working on the programs, there was no discussion about the documentation for their program assignments. In addition, every example program discussed in class was presented both as a literate program and as a traditional program, with the same amount of documentation in both versions.

Results: Two filter programs and the UNIX utility wc were used to obtain line, word, and character counts for the students programs. One filter program extracted the C source code without comments and the other filter program extracted the comments. Line, word, and character counts for the source code and comments for each program were obtained using wc. Tables 1 and 2 give the average line, word, and character counts for each group for program assignments 1 and 2 respectively. Recall that Group A did assignment 1 in AOPS and assignment 2 in Turbo C while Group B did assignment 1 in Turbo C and assignment 2 in AOPS. Notice that the source code and comment lines for each assignment are nearly identical, but the group using AOPS had substantially more comment words and comment characters.

To compare the two methods (literate programming and traditional) we used the ratio of comment to source lines, words and characters (see Table 3). An ANOVA showed a significant difference (p = 0.01) between the two methods for both the ratio of comments to source code words and the ratio of comment to source code characters.

	Source Code			Comments		
	Lines	Words	Chars	Lines	Words	Chars
Group A	285	729	4409	279	1706	10,673
Group B	298	785	4620	275	1277	8,686

Table 1. Counts for Programming Assignment No. 1

	Source Code			Comments		
	Lines	Words	Chars	Lines	Words	Chars
Group A	236	617	3856	219	984	6,424
Group B	230	619	3885	226	1418	9,223

Table 2. Counts for Programming Assignment No. 2

Comment/Source Code			
	Lines	Words	Characters
Literate	0.98	2.32	2.52
Traditional	0.93	1.61	1.78

Table 3. Ratio of Comment to Source Code Lines, Words and Characters.

It was surprising that the literate programs contained significantly more comment words and characters, but not more comment lines than the traditional programs. Table 4 shows a detailed analysis of the non-blank comment lines for the AOPS and Turbo C programs. A Mann-Whitney U test showed a significant difference ($p<0.001$) between the two methods for the number of comments, lines per comment, words per comment, and characters per comment. A major reason for the differences is that the traditional programs contained many marker comments such as /*end of while*/ that occupied part of a line. Whereas the literate program comments were usually in paragraphs and occupied entire lines.

To discover what the "more" documentation in literate programs is made up of, an analysis was done in which each comment in both groups was examined and a decision was made on whether the comment included a what element, a how element, or an example. A comment contains a what element if it describes the purpose of its associated block of code; it contains a how element if it describes the algorithm used by its associated block of code; and it contains an example if it gives an illustration of what its associated block of code does. The comment was also checked against its associated block of code to see if it was inconsistent. (An inconsistent comment is a comment that does not accurately describe its associated piece of code.) The data is summarized in Table 5.

Non-Blank Line Comments				
	Number	Lines	Words	Chars
Literate	31	253	1,562	9,948
Traditional	67	247	1,130	7,554

Table 4. Data for non-blank comment lines.

Number of comments that				
	include what	include how	include example	are inconsistent
Literate	all (512)	25	5	0
Traditional	all (1072)	0	0	0

Table 5. Analysis of Information Contents of Comments

All the comments examined in both groups were deemed to contain the what element. For literate programs, 25 comments (5%) were found to contain the how element and 5 comments (1%) were found to contain an example. No comment in the traditional group contained the how element or an example. This finding was very surprising since both groups were using the same commenting guidelines and grading scheme for both assignments. That both the literate and traditional versions contained "what comments" follows from the grading scheme. What is surprising is that only the literate versions contained "how comments" and examples. This suggests that literate programming inspires more substantial comments than traditional programming.

Although no inconsistencies were found between the internal comments and the code in both versions, we did find inconsistencies between the source code and external documentation in the traditional programs. For the traditional programs the students had to hand in a pseudocode design for their programs. The pseudocode design was embedded in the AOPS program. When we compared the pseudocode design and the source code for the Turbo C programs we found 10 instances of missing steps, extra steps, misspelled names, etc. We did not find any inconsistencies in the AOPS programs. For more information about this experiment, please refer to [11].

Although this experiment showed that the literate programs contain more documentation and better documentation than the traditional programs, a survey of general opinions about literate programming drew mixed opinions. When asked whether they would use a literate programming system to develop their own software, 13 out of 16 students responded yes. When asked why they would use a literate programming system, the majority responded that it was because the pseudocode was embedded in the source code which made the design step a logical step to do. When asked if they would use AOPS again, only 5 out of the 16 said they would. When asked why they would not use AOPS, the majority said it was too confusing and difficult to use AOPS when debugging a program.

Conclusion

Our experiment shows the literate programming paradigm indeed encourages more and consistent documentation. The students seem to like the literate programming style, except they do not like the debugging process required by the literate programming paradigm. However, this may be a legacy of Knuth's belief that in literate programming the source program should not be changed without changing the associated documentation. He purposely wanted the source code to be difficult to read so that the source file would be changed. The net result was that it was more likely that the documentation would be updated when a change was made to the program source code.

This seems to hint that the state of debugging in current literate programming systems is less than desirable. Although AOPS is language independent, it still could not escape the fact that debugging using a literate programming system is troublesome. Perhaps tools such as a debugger should be made as an integral part of a literate programming system.

Finally, a version of AOPS program including the the processor AOP, the lister AOL, and the hypertext browser AOB have been implemented for the IBM-PC using Turbo C. We will gladly furnish a copy of the AOPS program to anyone who sends us a diskette.

References

1. Avenarius, A. and Oppermann, S., "FWEB: A Literate Programming System For FORTRAN8x", *ACM SIGPLAN Notices,* Vol.25, No.1, pp. 52-58, Jan 1990.

2. Brown, M., and Cordes, D., "A Literate Programming Design Language", *Proc. CompEuro 90, IEEE International Conf. Computer Systems and Software Engineering*, IEEE CS Press, Los Alamitos, Calif., 1990, pp. 548-549.

3. Chapin, N., "Software Maintenance: A Different View", *AFIPS Conference Proceeding, 54th National Computer Conference,* 1985, pp. 509-513.

4. Fletton N., and Munro, M. "Redocumenting Software Systems Using Hypertext Technology", *IEEE Conference on Software Maintenance*, 1988, pp. 54-59.

5. Gurari, E. and Wu, J., "A WYSIWYG Literate Programming System [Preliminary Report]",*Proceedings CSC '91*, 1991, pp. 94-104.

6. Hyman, M., "Literate C++", *Computer Language*, Jul 1990, Vol. 7, No.7, pp. 67-79.

7. Knuth, D. E., "Literate Programming", *The Computer Journal*, Vol. 27, No. 2, 1984, pp. 97-111.

8. Martin, J. and McClure, C., *Software Maintenance: The Problem and Its Solutions.* Prentice-Hall, 1983.

9. Y. Nakamoto, Y., Iwamoto, T., Hori, M., Hagihara, K., Tokura, N., "An Editor for Documentation in C-system to Support Software Development and Maintenance", *IEEE 6th International Conference on Software Engineering*, 1982, pp. 330-339.

10. Reenskaug, T., Skaar A., "An Environment For Literate Smalltalk Programming", *OOPSLA 1989 Proceedings*, pp. 337-345.

11. Shum, S., Cook, C., "AOPS: An Abstraction Oriented Programming System For Literate Programming", *Software Engineering Journal* , Vol. 8 No. 3 (May 1993), pp. 113-120.

12. Standish, T., "An Essay on Software Reuse", *IEEE Trans on Software Engineering*, Vol.SE-10, No.5, pp. 494-497, Sep 1984.

13. Thimbleby, H., "Experience of 'Literate Programming' Using CWEB [a Variant of Knuth's WEB]", *The Computer Journal*, Vol. 29, No. 3, 1986, pp. 201-211.

14. Tung, S. H., "A Structured Method For Literate Programming", *Structured Programming*, 1989, Vol. 10, No. 2, pp.113-120.

15. Wu,Y. and Baker,T., "A Source Code Documentation System For Ada", *ACM Ada Letters*, Vol.9, No.5, pp. 84-88, Jul/Aug 1989.

16. Zvegintzov, N., "Nanotrends", *Datamation*, Aug 1983, pp. 106-116.