Literate Programming on a Team Project*

Norman Ramsey† and Carla Marceau

Odyssey Research Associates
301A Harris B. Dates Drive
Ithaca, New York 14850

February 4, 1991

Abstract

We used literate programming on a team project to write a 33,000-line program for the Synthesizer Generator. The program, Penelope, was written using `WEB`, a tool designed for writing literate programs. Unlike other `WEB` programs, many of which have been written by `WEB`'s developer or by individuals, Penelope was not intended to be published. We used `WEB` in the hope that both our team and its final product would benefit from the advantages often attributed to literate programming. The `WEB` source served as good internal documentation throughout development and maintenance, and it continues to document Penelope's design and implementation. Our experience also uncovered a number of problems with `WEB`.

Introduction

Donald Knuth coined the term "literate programming" when describing `WEB`, the tool he used to build TeX [1]. He believes that "the time is ripe for significantly better documentation of programs, and that we can best achieve this by considering programs to be *works of literature*." Knuth and others have presented examples of such programs [2, 3, 4, 5].

Literate programming is usually discussed in the context of publishing programs or of publishing books or articles about programs. TeX and METAFONT, the original applications of `WEB`, have been published as books [6, 7]. Other

*This research has been sponsored in part by the USAF, Rome Air Development Center, under contract number F30602–86–C–0071. The first author gratefully acknowledges the support of the Fannie and John Hertz Foundation.

†Current address: Department of Computer Science, Princeton University, Princeton, New Jersey 08544.

published literate programs seem to be primarily for teaching. Programs to find random sequences and count common words were written to illustrate the power of literate programming [2, 3]. Another program to count common words is a tutorial on how to develop and tune a small program [4]. Another illustrates formal methods of program development and the use of abstract data types [5].

A valedictory assessment points out three common aspects of the published literate programs: cosmetics, polish, and verisimilitude, of which verisimilitude—the property of using one input to produce both compilable program and published document–is deemed essential [8]. Additional expectations of a literate programming tool include flexible order of elaboration, ability to develop program and documentation concurrently in one place, cross-references, and indexing [9].

`WEB` is the principal tool used for literate programming; a number of implementations are available [1, 10, 11, 12, 13, 14]. `WEB` programmers interleave source code and descriptive text in a single document. When using `WEB`, a programmer divides the source code into small textual units called *modules*, and each module carries associated documentation. In the `WEB` source, different modules may be written in any order. The programmer is encouraged to choose an order that helps explain the program. The modules are like macro definitions; they have the form

`@<`*module name*`@>=`*body*

where the body contains both code and references to other modules. `WEB` programmers can also define and use macros similar to C preprocessor macros. One module is unnamed; its body represents the underlying program.

Two filters process `WEB` source. `TANGLE` reads the `WEB` source and expands the unnamed module according to its definition. References to named modules are expanded where they occur, and the end result is that `TANGLE` extracts the underlying program from the `WEB` source. This program is formed from the code fragments that the programmer put into the module definitions, assembled in the order required by the compiler. A second filter, `WEAVE`, reads the `WEB` source and converts it to TeX input, with which TeX can produce high-quality typeset documentation of the program. Examples of `WEB` source and typeset documentation can be found in Reference 1.

Using Literate Programming

Penelope is a language-based editor intended to help programmers develop formally verified Ada programs [15, 16, 17]. It parses annotated Ada programs, performs static semantic checking and overload resolution, computes weakest preconditions by predicate transformation, simplifies the resulting preconditions, and helps users construct proofs of those preconditions. Its source code is an editor specification for the Synthesizer Generator [18]. The Synthesizer Generator builds an editor of attributed syntax trees. When a user defines a tree, the editor attributes it. When the user changes the tree, the editor recomputes only those attribute values that have changed [19].

Penelope has two written specifications: the Ada language reference manual and a denotational-style definition of Ada predicate transformers [20, 21]. After three years of work, the `WEB` source for the editor is over 33,000 lines. Over 13,000 of those lines are interleaved documentation. The editor has been used to verify a software repeater for an asynchronous communication line and security properties of a part of the ASOS operating system kernel [22, 23, 24]. Seven programmers have written `WEB` source, but no more than four have worked on it at any one time. At this writing, the editor is being extended to support Ada libraries.

We decided to write the editor as a literate program because we expected implementing predicate transformation to be error-prone. To avoid errors, we used `WEB` to juxtapose the specification and implementation of each predicate transformer. We used TeX's math mode to write the specifications in the notation of denotational semantics. [21, 25].

The right model for a literate program that is being maintained and extended is not the novel but the car repair manual. We began writing new code as an explanation or tutorial for our colleagues, but as the text grew we treated it more like a reference work. As with any document, we revised repeatedly to clarify meaning. We often used `TANGLE`'s reordering mechanism to make major revisions in the `WEB` source without changing the underlying program.

We divided Penelope into parts according to functionality: we wrote chapters on abstract syntax, concrete syntax, predicate transformation, static semantic checking, proof construction, and simplification of terms. Each of the first four chapters follows the structure of the Ada Language Reference Manual, since each is closely related to Ada. Only knowledge of the Ada manual is required to navigate this code; a reader who knows where to find the `exit` statement in the Ada manual can find the predicate transformation of the `exit` statement in the corresponding section of the predicate transformation chapter. We gave the final chapters structures related to the problems of simplification and proof construction.

Penelope has grown so large that each chapter is itself structured like a reference work. Sections within chapters have a different structure. They are tutorials that use the traditional approach to documentation: begin with a problem statement or specification, discuss possible solutions, explain the design of the preferred solution, then develop the implementation and its description. A short introductory chapter describes the major parts of the program the organization used to form the text. A "how to find it" section helps readers find source code.

We have used the Penelope source not only for reference but also to introduce new programmers to the project.

We used some of the cosmetic features of `WEB`; we found tables of variables, functions, and files all useful to describe the organization of the code. When working on Ada static semantic checking, we found it useful to include in the `WEB` source some fragments of relevant documents. The fragments included visibility and overloading rules from the Ada reference manual [20] and a presentation of the Ada type system derived from Reference 26.

Evaluating WEB

Using `WEB` without help from its developer uncovered a number of problems. Some problems relate to the criteria in References 8 and 9, but others do not.

We had no trouble with `WEB`'s fundamental mechanisms, the section and the named module. They provide verisimilitude and flexible order of elaboration. The small size of modules makes it feasible to develop code and documentation in one place using an ordinary text editor. The order of fragments is the same in the `WEB` source and in the published document, which simplifies polishing.

We found some of `WEB`'s cosmetic features inadequate, others superfluous. Cosmetics should include appropriate media for presenting programs: not just math and tables but also diagrams and figures [27, 3, 28]. Describing data structures was hampered by TEX's lack of support for diagrams and pictures. TEX does make it easy to use mathematical notation, which helped considerably in describing predicate transformation, the simplifier, and the proof system used in the proof constructor.

Programmers complained more about prettyprinting than about all other `WEB` problems combined. `WEAVE` ignores the programmer's choice of indentation and line breaks; it breaks and indents lines on the basis of the syntactic categories of tokens. (Programmers can force line breaks by inserting special control sequences like `@/` into their program text, but that's about all.) We spent too much time tinkering with prettyprinting, trying to make `WEAVE`'s output acceptable to everyone. We would have been better off with no prettyprinter, or with a prettyprinter that changed only the typographic treatment of program fragments without changing the placement of tokens on the page.

We used the index of identifiers during reviews only, usually to find function definitions and to locate code that had been misplaced.

`WEB` formats interleaved documentation using TEX, which some programmers already use for writing documents. `WEB` and TEX are integrated poorly. One cannot lift pieces of `WEAVE`'s output and put them in other documents without adjusting the TEX code. `WEB` works even less well with LATEX; LATEX constructs cannot be used in `WEB` source, and getting `WEAVE` output to work in LATEX documents requires tedious adjustments by hand. The other direction is easier; we wrote project documents using LATEX, and, with some adjustments, we could include excerpts from these documents in our code. We were accustomed to writing LaTeX documents, so it was annoying to be forced to switch to plain TeX to write programs.

TeX's input often looks very different from TeX's output, especially when mathematics is used. Similarly, `WEB` source looks very different from the typeset document produced by `WEAVE` and TeX. When people use TeX to write papers, it may be acceptable for the TeX input to be full of confusing hieroglyphics, because the TeX input is almost never read—only the printed version is read. `WEB` source is read frequently because programmers must edit it. Both the difficulty of reading the source and the marked difference between source and listing complicate editing.

`WEAVE`'s standard table of contents mechanism is a list of section names. (A section is a named group of modules together with their interleaved documentation.) This flat structure is inadequate for describing even a moderate-sized program. Extra structure in the form of "parts" has been added to the book version of TeX; a part contains several sections [6].

We changed `WEAVE`'s table of contents mechanism to make hierarchical organization possible. The new mechanism supports chapters, sections, and two levels of subsections. We did so without changing `WEAVE` itself; instead we changed the TeX macros that support `WEAVE`. The new macros recognize special symbols at the beginnings of section names; these symbols indicate which sections are really chapters, subsections, and so on. Using this hierarchy made the table of contents an important guide to the code; the only practical way to find a particular part of the editor code was to begin with the table of contents.

The `WEAVE` listing of Penelope is over 800 pages; its table of contents is about 8 pages. We needed to extract and print parts of this document, but `WEAVE` processes only complete documents. We extracted parts in two ways. When we wanted just a few, small, closely related parts, we created a special `WEB` file that held just those parts, and printed it. For something more general, we used a shell script that removed parts of `WEAVE`'s output before passing the rest to TeX. This script recognized the special symbols in the section names so that, for example, we could include or exclude whole chapters by name, without having to enumerate their contents. It would have been more expensive, and no less awkward, to use standard mechanisms for extracting pages from TeX's output.

Special symbols in section names are an awkward way of indicating structure. A more natural way of indicating structure, like the LaTeX mechanism, would have been welcome. Tools should make it easy to use the structure to help readers extract excerpts from literate programs.

`WEB` works poorly with `make` [29]. `TANGLE` is designed to read and write a complete program. Some `TANGLE`s can write multiple files, which can then be compiled separately, but those files all get rewritten every time the `WEB` source changes, and `make` therefore recompiles them all. This problem is familiar;

other preprocessors, like `yacc` [30], can also cause excessive recompilation. The workaround described on page 265 of Reference 31 works for `TANGLE`.

`WEB` users may be tempted to break their `WEB` source into many files and run them through `TANGLE` separately. Doing so defeats the purpose of writing a literate program; separate compilation does not necessarily imply separate explanation. For example, one would prefer to place a unit's specification and implementation in the same `WEB` source, even when they should be compiled separately.

Our `TANGLE` [32] emits the `#line` directive of the C preprocessor, making messages from compilers and debuggers refer to line numbers in the `WEB` source instead of to those in `TANGLE`'s output. Not all compilers or all `TANGLE`s support such mechanisms. Renumbering is essential for large programs.

Discussion

Most literate programming papers refer to programs that are polished, publishable "works of art." Our primary goal in writing Penelope was not to write a publishable program, nor to evaluate literate programming as a software engineering technique, but to build a prototype editor embodying the results of research in formal verification. Our experience has given us some subjective impressions of the benefits of applying literate programming to team development, as well as more specific conclusions about the difficulties of applying literate programming.

There are few published techniques for writing literate programs, especially for writing large ones. Most of our programmers complained about the awkward tools and about the lack of guidance in their use. We were unable to develop precise methods for writing literate programs or even clear criteria about what constituted a literate program. Instead, we relied on peer review of programs, rewriting them until the project members understood them. These programs were rarely polished to the point necessary for publication; less polished presentations were adequate.

One review of a literate program emphasizes the role of juxtaposed code and documentation [9]. It cites several benefits of this juxtaposition, including an incentive to explain and hence to understand what one is doing. During peer reviews of Penelope, we insisted that explanations of programs include explanations of design. Juxtaposing design documentation and code reduced the overhead of maintenance because maintainers reading the code did not have to look elsewhere for design documents.

We cannot say to what extent literate programming can replace standard software development methodology. However, putting a clear description of design in our source code helped make it possible for a changing team of programmers to develop it over a span of three years. We were able to apply standard guidelines, like those in Reference 33, to describing Penelope's design.

We believe literate programming helped us substantially. This belief is based not on measurements but on our subjective comparisons of experience on this project to other projects. A programmer who has used standard software development systems at an international computer manufacturing company reports that a key difference in Penelope was that the documentation was *used*, precisely because of its proximity to the source code. The project manager, when at a large software house, learned to expect technical staff first to criticize implementations for drifting away from the original intent, then to call for complete rewrites. There have been no complaints about the quality of Penelope's implementation. The programmers have been surprised at how easily they have extended and modified one another's work. For example, an editor for constructing proofs was implemented by a programmer who then left the project. The programmer who took over the job of maintaining the proof constructor read the program in two hours and found herself well prepared to change the code.

We do not ascribe Penelope's success purely to literate programming; other factors contributed. We began implementation with a clearly defined goal and worked from detailed, sometimes formal, specifications. We had time to design the system carefully and used a declarative programming language. We did not have to develop a user interface but used the one provided by the Synthesizer Generator [18].

We will continue to use literate programming for Penelope and for future projects. `WEB`'s problems are such that we believe it will be cost-effective for future projects to develop literate programming tools that address some of the criticisms presented here.

Acknowledgments

Silvio Levy provided his `CWEB` implementation as a base for the `WEB` used to implement the Penelope editor. The staff of the Ada Verification project at Odyssey Research made possible the experience with Penelope on which this paper is based. This paper has been much improved by comments from David Hanson and from the anonymous referees. Stuart Feldman commented on this paper and discussed ideas for better tools.

References

[1] Donald E. Knuth. Literate programming. *The Computer Journal*, 27(2):97–111, 1984.

[2] Donald E. Knuth and Jon L. Bentley. Programming pearls: Literate programming. *Communications of the ACM*, 29(5):364–368, May 1986.

[3] Donald E. Knuth. Programming pearls: A literate program. *Communications of the ACM*, 29(6):471–483, June 1986. Reviewed by M. Douglas McIlroy.

[4] David R. Hanson. Literate programming: Printing common words. *Communications of the ACM*, 30(7):593–599, July 1987. Reviewed by John Gilbert.

[5] David Gries and Jon Bentley. Programming pearls: Abstract data types. *Communications of the ACM*, 30(4):284–290, April 1987.

[6] Donald E. Knuth. *TEX: The Program*, volume B of *Computers & Typesetting*. Addison-Wesley, 1986.

[7] Donald E. Knuth. METAFONT*: The Program*, volume D of *Computers & Typesetting*. Addison-Wesley, 1986.

[8] Christopher J. Van Wyk. Literate programming: An assessment. *Communications of the ACM*, 33(3):361–365, March 1990.

[9] Harold Thimbleby. A review of Donald C. Lindsay's text file difference utility, *diff*. *Communications of the ACM*, 32(6):752–755, June 1989.

[10] Harold Thimbleby. Experiences of 'literate programming' using cweb (a variant of Knuth's WEB). *Computer Journal*, 29(3):201–211, 1986.

[11] Klaus Guntermann and Joachim Schrod. `WEB` adapted to C. *TUGboat*, 7(3):134–137, October 1986.

[12] Silvio Levy. `WEB` adapted to C, another approach. *TUGBoat*, 8(1):12–13, April 1987.

[13] Wayne Sewell. How to `MANGLE` your software: the `WEB` system for Modula-2. *TUGboat*, 8(2):118–128, July 1987.

[14] Norman Ramsey. Literate programming: Weaving a language-independent `WEB`. *Communications of the ACM*, 32(9):1051–1055, September 1989.

[15] Norman Ramsey. Developing formally verified Ada programs. In *Proceedings of the 5th International Workshop on Software Specification and Design*, pages 257–265, Pittsburgh, Pennsylvania, May 1989.

[16] Carla Marceau and C. Douglas Harper. An interactive approach to Ada verification. In *Proceedings of the 12th National Computer Security Conference*, pages 28–51, Baltimore, Maryland, October 1989.

[17] David Guaspari, Carla Marceau, and Wolfgang Polak. Formal verification of Ada programs. *IEEE Transactions on Software Engineering*, 16(9):1058–1075, September 1990.

[18] Thomas Reps and Tim Teitelbaum. *The Synthesizer Generator Reference Manual.* Springer-Verlag, 1989.

[19] Thomas Reps. *Generating Language-Based Environments.* MIT Press, 1984.

[20] US Department of Defense. *The Ada Programming Language Reference Manual*, 1983. ANSI/MILSTD 1815A.

[21] Wolfgang Polak. Predicate transformer semantics for Ada. Technical Report 89-39, Odyssey Research Associates, September 1989.

[22] Carla Marceau and Geoffrey Hird. A verified software implementation of an RS-232 repeater using Penelope. Technical Report 90-12, Odyssey Research Associates, 1990.

[23] D. G. Weber and Roger L. Costello. Beyond A1 using Ada code verification. Technical Report 89-9, Odyssey Research Associates, April 1989.

[24] Eric R. Anderson, Ben DiVitto, and Ruth M. Hart. ASOS: Information security for real-time systems. In *AFCEA West Intelligence Symposium*, 1987.

[25] Wolfgang Polak. Program verification based on denotational semantics. In *ACM Symposium on Principles of Programming Languages*, pages 149–158. Association for Computing Machinery, 1981.

[26] H. Ganzinger and K. Ripken. Operator identification in Ada. *ACM SIGPLAN Notices*, 15(2):30–42, February 1980.

[27] Jon Bentley. *More Programming Pearls: Confessions of a Coder*, chapter 10 and 11, pages 101–126. Addison-Wesley, 1988.

[28] Robert Sedgewick. *Algorithms.* Addison-Wesley, second edition, 1988.

[29] Stuart I. Feldman. Make—a program for maintaining computer programs. *Software—Practice and Experience*, 9:255–265, 1979.

[30] Steve C. Johnson. Yacc—yet another compiler compiler. Technical Report 32, Computer Science, AT&T Bell Laboratories, Murray Hill, New Jersey, 1975.

[31] Brian W. Kernighan and Rob Pike. *The UNIX Programming Environment.* Prentice-Hall, Englewood Cliffs, NJ, 1984.

[32] Norman Ramsey. The Spidery `WEB` system of structured documentation. Technical Report TR-226-89, Department of Computer Science, Princeton University, August 1989.

[33] David Lorge Parnas and Paul C. Clements. A rational design process: How and why to fake it. *IEEE Transactions on Software Engineering*, SE-12(2):251–257, February 1986.