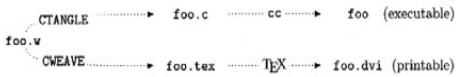

Donald Knuth. "Literate Programming (1984)" in Literate Programming. CSLI, 1992, pg. 99.

I believe 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. Hence, my title: "Literate Programming."

Let us change our traditional attitude to the construction of programs: Instead of imagining that our main task is to instruct a computer what to do, let us concentrate rather on explaining to human beings what we want a computer to do.

The practitioner of literate programming can be regarded as an essayist, whose main concern is with exposition and excellence of style. Such an author, with thesaurus in hand, chooses the names of variables carefully and explains what each variable means. He or she strives for a program that is comprehensible because its concepts have been introduced in an order that is best for human understanding, using a mixture of formal and informal methods that reinforce each other.

Ross Williams. FunnelWeb Tutorial Manual, pg 4.

A traditional computer program consists of a text file containing program code. Scattered in amongst the program code are comments which describe the various parts of the code.

In literate programming the emphasis is reversed. Instead of writing code containing documentation, the literate programmer writes documentation containing code. No longer does the English commentary injected into a program have to be hidden in comment delimiters at the top of the file, or under procedure headings, or at the end of lines. Instead, it is wrenched into the daylight and made the main focus. The "program" then becomes primarily a document directed at humans, with the code being herded between "code delimiters" from where it can be extracted and shuffled out sideways to the language system by literate programming tools.

The effect of this simple shift of emphasis can be so profound as to change one's whole approach to programming. Under the literate programming paradigm, the central activity of programming becomes that of conveying meaning to other intelligent beings rather than merely convincing the computer to behave in a particular way. It is the difference between performing and exposing a magic trick.

Wayne Sewell. "Integral Pretty-printing" in Weaving a Program: Literate Programming in WEB, Van Nostrand Reinhold, 1989, pg. 7.

Listings generated by the WEB system are unlike any other form of program listings in existence. They resemble programs from computer science textbooks rather than listings from executable programs. WEB utilizes the TeX document compiler, which includes a typesetting command language capable of tremendous control over document appearance. Even if the author of a WEB program does not directly utilize TeX capabilities in the source code, the combined efforts of WEB and TeX will create beautiful documents on their own. TeX automatically handles details such as microjustification, kerning, hyphenation, ligatures, and other sophisticated operations, even when the description part of the source is simple ASCII text. WEB adds functions which are specific to computer programs, such as boldface reserved words, italicized identifiers, substitution of true mathematical symbols, and more standard pretty-printer functions such as reformatting and indentation.

Wayne Sewell. "Reduction of Visual Complexity" in Weaving a Program: Literate Programming in WEB, Van Nostrand Reinhold, 1989, pg. 42.

The whole concept of code sections, indeed structured programming, is to reduce the amount of text that must be read in order to determine what a piece of code is doing. The code section is a form of data reduction in that the section name is a placeholder representing the code contained in that section. Anything that is logically part of the section should be moved into it, thereby reducing the complexity of the code where it is referenced.

Bart Childs. "Literate Programming, A Practitioner's View", Tugboat, December 1992, pg. 261-262.

I use the following list of requirements to imply a definition of a literate program and the minimum set of tools which are needed to prepare, use, and study the resulting code.

- The high-level language code and the system documentation of the program come from the same set of source files.
- The documentation and high-level language code are complementary and should address the same elements of the algorithms being written.
- The literate program should have logical subdivisions. Knuth called these modules or sections.
- The system should be presented in an order based upon logical considerations rather than syntactic constraints.
- The documentation should include an examination of alternative solutions and should suggest future maintenance problems and extensions.

Donald Knuth. The CWEB System of Structure Documentation. Addison-Wesley. 1994. pg. 1.

The philosophy behind CWEB is that an experienced system programmer, who wants to provide the best possible documentation of his or her software products, needs two things simultaneously: a language like TeX for formatting, and a language like C for programming. Neither type of language can provide the best documentation by itself; but when both are appropriately combined, we obtain a system that is much more useful than either language separately.

The structure of a software program may be thought of as a "WEB" that is made up of many interconnected pieces. To document such a program we want to explain each individual part of the web and how it relates to its neighbors. The typographic tools provided by TeX give us an opportunity to explain the local structure of each part by making that structure visible, and the programming tools provided by languages like C make it possible for us to specify the algorithms formally and unambiguously. By combining the two, we can develop a style of programming that maximizes our ability to perceive the structure of a complex piece of software, and at the same time the documented programs can be mechanically translated into a working software system that matches the documentation.

Besides providing a documentation tool, CWEB enhances the C language by providing the ability to permute pieces of the program text, so that a large system can be understood entirely in terms of small sections and their local interrelationships. The CTANGLE program is so named because it takes a given web and moves the sections from their web structure into the order required by C; the advantage of programming in CWEB is that the algorithms can be expressed in "untangled" form, with each section explained separately. The CWEAVE program is so named because it takes a given web and intertwines the TeX and C portions contained in each section, then it knits the whole fabric into a structured document.

John Krommes.

The fundamental logic of the WEB system encourages "top-down" programming and "structured" design. Quoting from Kernighan and Plauger, 'Top-down design and successive refinement attack a programming task by specifying it in the most general terms, then expanding these into more and more specific and detailed actions, until the whole program is complete. Structured design is the process of controlling the overall design of a system or program so the pieces fit together neatly, yet remain sufficiently decoupled that they may be independently modified. ... Each of these disciplines can materially improve programmer productivity and the quality of code produced.' The WEB system encourages you to work top-down by giving you the ability to break up your code into independent segments (called "sections").

Doug McIlroy. "Programming Pearls: A Literate Program", CACM, June 1986, pg. 478-479.

The presentation is engaging and clear. In WEB one deliberately writes a paper, not just comments, along with code. This of course helps readers. I am sure that it also helps writers: reflecting upon design choices sufficiently to make them explainable must help clarify and refine one's thinking. Moreover, because an explanation in WEB is intimately combined with the hard reality of implementation, it is qualitatively different from, and far more useful than, an ordinary "specification" or "design" document. It can't gloss over the tough places.

Matt Pharr and Greg Humphries. "Physically Based Rendering: From Theory to Implementation", Morgan Kaufmann, 2004

Writing a literate program is a lot more work than writing a normal program. After all, who ever documents their programs in the first place!? Moreover, who documents them in a pedagogical style that is easy to understand? And finally, who ever provides commentary on the theory and design issues behind the code as they write the documentation? All of that is here in the pages that follow.

This book presents a selection of modern rendering algorithms through the documented source code for a complete rendering system. The system, pbrt, is written using a programming methodology called literate programming that mixes prose describing the system with source code that implements it. We believe that the literate programming approach is a valuable way to introduce ideas in computer graphics and computer science in general. Often, some of the subtleties of an algorithm can be unclear or hidden until it is implemented, so seeing an actual implementation is a good way to acquire a solid understanding of that algorithm's details. Indeed we believe that deep understanding of a small number of algorithms in this manner provides a stronger base for further study of computer graphics than does a superficial understanding of many.

This book is a long literate program. This means that in the course of reading this book, you will read the full implementation of the pbrt rendering system, not just a high-level description of it. The literate programming metalanguage provides two important features. The first is the ability to mix prose with source code. This feature makes the description of the program just as important as its actual source code, encouraging careful design and documentation. Second, the language provides a mechanism for presenting program code to the reader in an entirely different order than it is supplied to the compiler. Thus the program can be described in a logical manner.

- The documentation should include a description of the problem and its solution. This should include all aids such as mathematics and graphics that enhance communication of the problem statement and the understanding of its challenge.
- Cross references, indices, and different fonts for text, high-level language keywords, variable names, and literals should be reasonably automatic and obvious in the source and the documentation.

WEB's design encourages writing programs in small chunks which Knuth called modules (he also used the term sections). Modules have three parts: documentation, definitions, and code. At least one of these three parts must be non-null.

The documentation portion is often a verbal description of the algorithm. It may be any textual information that aids the understanding of the problem. It is not uncommon for a WEB to have a number of `documentation only' modules. These usually describe the problem independent of the chosen language for implementation. For example, a WEB for a subprogram that solves the linear equation, Ax = b, could have discussion of singularity, condition numbers, partial pivoting, the banded nature of the expected coefficient matrices, etc. It should be an unusual but not exceptional case when a module contains no documentation.

Daniel Mall. "Recommendation for Literate Programming"

The key features of literate programming are the organization of source code into small sections and the production of a book quality program listing. Literate programming is an excellent method for documenting the internals of software products especially applications with complex features. Literate programming is useful for programs of all sizes. Literate programming encourages meaningful documentation and the inclusion of details that are usually omitted in source code such as the description of algorithms, design decisions, and implementation strategy.

Literate programming increases product quality by requiring software developers to examine and explain their code. The architecture and design is explained at a conceptual level. Modeling diagrams are included (UML). Long procedures are restructuring by folding portions of the code into sections. Innovative ideas, critical technical knowledge, algorithmic solutions, and unusual coding constructions are clearly documented.

Literate programs are written to be read by other software developers. Program comprehension is a key activity during corrective and perfective maintenance. High quality documentation facilitates program modification with fewer conceptual errors and resultant defects. The clarity of literate programs enables team members to reuse existing code and to provide constructive feedback during code reviews.

Organization of source code into small sections. The style of literate programming combines source code and documentation into a single source file. Literate programs utilize sections which enable the developer to describe blocks of code in a convenient manner. Functions are decomposed into several sections. Sections are presented in the order which is best for program comprehension. Code sections improve on verbose commenting by providing the ability to write descriptive paragraphs while avoiding cluttering the source code.

Production of a book quality program listing. Literate programming languages (CWEB) utilize a combination of typesetting language (TeX) and programming language (C++). The typesetting language enables all of the comprehension aids available in books such as pictures, diagrams, figures, tables, formatted equations, bibliographic references, table of contents, and index. The typographic processing of literate programs produces code listings with elegantly formatted documentation and source code. Listings generated in PDF format include hypertext links.

Remember the Basics. There are many factors involved in developing excellent software. Literate programming is just a single technique to be used along with all the other well established software engineering practices. Here are some software practices related to program documentation:

- Establish structures, processes, and outcomes (see Luke Holman).
- Generate software requirements and design description (see IEEE standards).
- Practice object oriented design.
- Choose class names, function names, and variable names wisely.
- Avoid duplicate code by creating shared functions.
- Re-think or refactor code which is difficult to understand.
- Develop small classes and small functions when feasible.
- Keep it simple and straight forward as much as possible.
- Organize large source code files using an outlining editor (Leo).
- Comment source code effectively with header and in-line comments.
- Document source code using an API documentation standard (doxygen).
- Utilize pre-conditions and post-conditions using assertions.
- Provide formal or informal proofs of source code correctness.
- Conduct peer reviews of deliverables.
- Implement automated unit testing which is also a form of documentation.
- Examine source code metrics (lines, complexity, etc).
- Execute static analysis for common coding errors.

Some of my favorite tools are CWEB and Leo for source code outlining, doxygen for API documentation, CCCC and LocMetrics for source code metrics, PC Lint for static error analysis, Response Time Tracker for algorithm performance analysis, and cppunit for automated unit testing.

In some sense, the literate programming system is just an enhanced macro substitution package tuned to the task of rearranging source code. This may seem like a trivial change, but in fact literate programming is quite different from other ways of structuring software systems.

John Gilbert. "Literate Programming: Printing Common Words", CACM, July 1987, pg 599.

Literacy in programming means different things in different circumstances. It's not a matter of artistry or efficiency alone; it's more a question of suitability in context. Knuth's expository gem will teach future readers about programming style and data structures, whether they use the code or not. McIlroy's six liner is not itself an enduring piece of work, but it is a clear example of how to use enduring tools. Hanson's real-world code, then, must be evaluated according to whether it is robust, flexible, and easy to maintain. Despite roughness in low-level style, the program meets these goals well. Hanson demonstrates that "literate programming" is a viable approach to creating works of craft as well as works of art.

Marc van Leeuwen. "Requirements for Literate Programming" in CWEBx Manual, pg. 3-4.

The basic idea of literate programming is to take a fundamentally different starting point for the presentation of programs to human readers, without any direct effect on the program as seen by the computer. Rather than to present the program in the form in which it will be compiled (or executed), and to intercalate comments to help humans understand what is going on (and which the compiler will kindly ignore), the presentation focuses on explaining to humans the design and construction of the program, while pieces of actual program code are inserted to make the description precise and to tell the computer what it should do. The program description should describe parts of the algorithm as they occur in the design process, rather than in the completed program text. For reasons of maintainability it is essential however that the program description defines the actual program text; if this were defined in a separate source document, then inconsistencies would be almost impossible to prevent. If programs are written in a way that concentrates on explaining their design to human readers, then they can be considered as works of (technical) literature; it is for this reason that Knuth has named this style of software construction and description "literate programming".

The documentation parts of the program description should allow for the same freedom of expression that one would have in an ordinary technical paper. This means that the document describing the program should consist of formatted text, rather than being a plain text file. This does not exclude the possibility that the source is written as a plain text file, but then it should undergo some form of processing to produce the actual program description. The document should moreover contain fragments of a program written in some traditional (structured) programming language, in such a way that they can be mechanically extracted and arranged into a complete program; in the formatted document on the other hand layout and choice of fonts for these program fragments should be so as to maximize readability.

Parts of the program that belong together logically should appear near to each other in the description, so that they are visible from the part of the documentation that discusses their function. This means that it should be possible to rearrange program text with respect to the order in which it will be presented to the computer, for otherwise the parts that deal with the actions at the outer level of a subroutine will be pushed apart by the pieces specifying the details of inner levels. The most obvious and natural way to do this is to suppress the program text for those inner levels, leaving an outline of the outer level, while the inner levels may be specified and documented elsewhere; this is a bit like introducing subroutines for the inner levels, but without the semantic implications that that would have. There should be no restrictions on the order in which the program fragments resulting from this decomposition are presented, so that this order can be chosen so as to obtain an optimal exposition; this may even involve bringing together fragments whose location in the actual program is quite unrelated, but which have some logical connection.

Obviously there should be a clear indication of where pieces of program have been suppressed, and which other program fragments give the detailed specifications of those pieces. From the programming language point of view the most obvious method of identification would be to use identifiers, resulting in a simple system of parameter-less macros, with as only unusual aspect that uses of the macro are allowed to precede the definition, and indeed do so more often than not. Actually, literate programming uses a method that differs from this only trivially from a formal standpoint, but has a great advantage in practical terms: identification is by means of a more or less elaborate phrase or sentence, marked in a special way to indicate that it is a reference to a program fragment. This description both stands for the fragment that is being specified elsewhere, and also serves as a comment describing the function of that fragment at a level of detail that is appropriate for understanding the part of the program containing it. In this way several purposes are served at once: a clear identification between use and definition is established, the code at the place of use is readable because irrelevant detail is suppressed, with a relevant description of what is being done replacing it, and at the place of definition a reminder is given of the task that the piece of code presented is to perform. The documenting power of such a simple device is remarkable. In some cases the result is so clear that there is hardly any need to supply further documentation; also it can sometimes be useful to use this method to replace small pieces of somewhat cryptic code by a description that is actually longer than the code itself.

Mark Wallace. The Art of Donald E. Knuth

If his attention to the minutiae of programming has earned the annoyance of a younger generation of programmers, though, Knuth remains the éminence grise of algorithm analysis, and one of the leading thinkers on programming in general.

Of course, other computer scientists have made contributions to the field that are every bit as substantial (most notably Edsger Dijkstra, Charles Hoare and Niklaus Wirth). But Knuth's work brings to life the complex mathematical underpinnings of the discipline, and deals with the logistics of programming on all levels, from the conceptual design of solutions to the most

Donald Knuth. "Questions and Answers with Prof. Donald E. Knuth", CSTUG, Charles University, Prague, March 1996, pg. 227-229.

I was talking with Tony Hoare, who was editor of a series of books for Oxford University Press. I had a discussion with him in approximately 1980; I'm trying to remember the exact time, maybe 1979, yes, 1979, perhaps when I visited Newcastle? I don't recall exactly the date now. He said to me that I should publish my program for TeX. [I looked up the record when I returned home and found that my memory was gravely flawed. Hoare had heard rumors about my work and he wrote to Stanford suggesting that I keep publication in mind. I replied to his letter on 16 November 1977-much earlier than I remembered.]

As I was writing TeX I was using for the second time in my life ideas called "structured programming", which were revolutionizing the way computer programming was done in the middle 70s. I was teaching classes and I was aware that people were using structured programming, but I hadn't written a large computer program since 1971. In 1976 I wrote my first structured program; it was fairly good sized-maybe, I don't know, 50,000 lines of code, something like that. (That's another story I can tell you about sometime.) This gave me some experience with writing a program that was fairly easy to read. Then when I started writing TeX in this period (I began the implementation of TeX in October of 1977, and I finished it in May 78), it was consciously done with structured programming ideas.

Professor Hoare was looking for examples of fairly good-sized programs that people could read. Well, this was frightening. This was a very scary thing, for a professor of computer science to show someone a large program. At best, a professor might publish very small routines as examples of how to write a program. And we could polish those until ... well, every example in the literature about such programs had bugs in it. Tony Hoare was a great pioneer for proving the correctness of programs. But if you looked at the details ... I discovered from reading some of the articles, you know, I could find three bugs in a program that was proved correct. [laughter] These were small programs. Now, he says, take my large program and reveal it to the world, with all its compromises. Of course, I developed TeX so that it would try to continue a history of hundreds of years of different ideas. There had to be compromises. So I was frightened with the idea that I would actually be expected to show someone my program. But then I also realized how much need there was for examples of good-sized programs, that could be considered as reasonable models, not just small programs.

I had learned from a Belgian man (I had met him a few years earlier, someone from Liege), and he had a system-it's explained in my paper on literate programming. He sent me a report, which was 150 pages long, about his system-it was inspired by "The Ghost in the Machine". His 150-page report was very philosophical for the first 99 pages, and on page 100 he started with an example. That example was the key to me for this idea of thinking of a program as hypertext, as we would now say it. He proposed a way of taking a complicated program and breaking it into small parts. Then, to understand the complicated whole, what you needed is just to understand the small parts, and to understand the relationship between those parts and their neighbors. [Pierre Arnoul de Marneffe, Holon Programming. Univ. de Liege, Service d'Informatique (December, 1973).]

In February of 1979, I developed a system called DOC and UNDOC ... something like the WEB system that came later. DOC was like WEAVE and UNDOC was like TANGLE, essentially. I played with DOC and UNDOC and did a mock-up with a small part of TeX. I didn't use DOC for my own implementation but I took the inner part called getchar, which is a fairly complicated part of TeX's input routine, and I converted it to DOC. This gave me a little 20-page program that would show the getchar part of TeX written in DOC. And I showed that to Tony Hoare and to several other people, especially Luis Trabb Pardo, and got some feedback from them on the ideas and the format.

Then we had a student at Stanford whose name was Zabala-actually he's from Spain and he has two names-but we call him Inaki; Ignacio is his name. He took the entire TeX that I'd written in a language called SAIL (Stanford Artificial Intelligence Language), and he converted it to Pascal in this DOC format. TeX-in-Pascal was distributed around the world by 1981, I think. Then in 1982 or 1981, when I was writing TeX82, I was able to use his experience and all the feedback he had from users, and I made the system that became WEB. There was a period of two weeks when we were trying different names for DOC and UNDOC, and the winners were TANGLE and WEAVE. At that time, we had about 25 people in our group that would meet every Friday. And we would play around with a whole bunch of ideas and this was the reason for most of the success of TeX and METAFONT.

Pierre-Arnoul de Marneffe. Holon Programming: A Survey, 1973, pg. 8-9.

The "Holon" concept has been introduced in biological and behavior sciences by Koestler. This concept proceeds from the work of Simon. It is used for instance to analyze complex living organisms or complex social systems. This neologism is from Greek "holos", i.e., whole, and the suffix "-on" meaning "part". A holon is a "part of a whole". The reader is forewarned to not mix up the holon concept with the "module" one.

"Hierarchy": Each holon is composed by other holons which are "refinements" of the former holon. These holons are submitted to some rigid rules; they perform the "detail" operations which, put together, compose the function of the former holon.

"Tendency to Integration": The holon integrates with other holons in the hierarchy according to a flexible strategy. This integration must be understood as a will to close cooperation with the other holons for the emergence of a "tougher" and more efficient component.

intimate details of the machine. The fundamental elements of any computer program are, perhaps not surprisingly, time and space. (In programming terms, time describes the speed with which a program accomplishes its task, while space refers to the amount of memory a program requires both to store itself -- i.e. the length of the code -- and to compute and store its results.) But Knuth is concerned not only with bytes and microseconds, but with a concept that has come to be known in coding circles as "elegance," and that applies to programming at any level.

Elegance takes in such factors as readability, modular coding techniques and the ease with which a program can be adapted to other functions or expanded to perform additional tasks. (Knuth's broader ideas about documentation and structured programming are laid out in his 1992 book, "Literate Programming.") Though rarely mentioned, "sloppy coding" often costs companies a great deal in terms of time and money; programmers brought in to update the code of consultants gone by must spend hours or days deciphering a poorly documented program, or hunting down bugs that might have been caught easily had the initial programmer simply been a bit more conscientious in the practice of his craft.

Besides demonstrating the techniques of clear, efficient coding, Knuth has sought to bring a deeper sense of aesthetics to the discipline. "You try to consider that the program is an essay, a work of literature," he says. "I'm hoping someday that the Pulitzer Prize committee will agree." Prizes would be handed out for "best-written program," he says, only half-joking. Knuth himself has already collected numerous awards, including the National Medal of Science from then-President Jimmy Carter and Japan's prestigious Kyoto Prize.

Gregory Wilson. "XML-Based Programming Systems", Dr. Dobbs Journal, March 2003, pg. 16-24.

Many programming environments are completely controlled by specific vendors, who may well choose to switch from flat text to rich markup for their own reasons. If Microsoft had made source files XML, tens of thousands of programmers would already be putting pictures and hyperlinks in their code. Programming on the universal canvas is one revolution that can't possibly arrive too soon.

Astonished Reader. "On Misreadings", email, January 2009.

Read your first page: YOU GOT IT TOTALLY WRONG. Literate programming is NOT about documentation in the first place. All quotes you tore out speak of literate programming as if it's just a documentation system. While it is not.

Literate programming is a PROGRAMMING PARADIGM, or if you wish a "META-LANGUAGE", on top of machine-coding language, which was created with the purpose of: a) allowing humans to create abstractions over abstractions over abstractions with macros (which are phrases in a human language and if you wish are precise "new operators" in that meta-language, created on the fly). b) this system of macros can be created not in machine demanded order, but as need for logical thinking. Later it is reshuffled ("tangled", i.e. convoluted, scrambled) from the natural into the inhuman machine codes.

You totally missed the idea, and in the case of blind leading the blind quote scores of other misreaders. Literate programming is not a documentation system per ce, it's a programming paradigm.

Maurice V. Wilkes, David J. Wheeler, and Stanley Gill. The Preparation of Programs for an Electronic Digital Computer, with special reference to the EDSAC and the use of a library of subroutines. Tomash Publishers (reprint series), 1951, pg. 22.

3-1 Open subroutines. The simplest form of subroutine consists of a sequence of orders which can be incorporated as it stands into a program. When the last order of the subroutine has been executed the machine proceeds to execute the order in the program that immediately follows. This type of subroutine is called an "open" subroutine.

3-2 Closed subroutines. A "closed" subroutine is one which is called into use by a special group of orders incorporated in the master routine or main program. It is designed so that when its task is finished it returns control to the master routine at a point immediately following that from which it was called in. The subroutine itself may be placed anywhere in the store.