www.literateprogramming.com

Literate Programming

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.

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.

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.

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 commentary, doxygen for API documentation, CCCC for source code metrics and LocMetrics for source code metrics, PC Lint for static error analysis, and cppunit for automated unit testing. My career is performance analyst for Parzall.

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.

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 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").

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.
- 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.

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.

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.

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 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.

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.

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.

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.

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.

Software Documentation

Steve McConnell. Code Complete. Microsoft Press, 1993, pg. 453.

Most programmers enjoy writing documentation if the documentation standards aren't unreasonable. Like layout, good documentation is a sign of the professional pride a programmer puts into a program.

Mike Punter. "Programming for Maintenance" in Techniques of Program and System Maintenance. QED Information Sciences, 1988, pg. 116.

- Sooner or later someone else is going to have to understand the programs you write.
- Programs must be written for people as well as computers.
- Knowing that a program can be understood and amended by someone else ought to be one
  of the programmer's criteria for success.

Robert Riggs. "Computer Systems Maintenance" in Techniques of Program and System Maintenance. QED Information Sciences, 1988, pg. 145.

I mention good documentation as the first requirement because of its overriding importance. If you run a sloppy systems and programming shop, you might as well forget the organization and management of a systems maintenance group and just hope that the authors stay around. In fact, a very high percentage of production programs must be maintained by a programmer other than the original author. Failure to complete good documentation puts you in a vicious, never ending circle of crises.

Tom DeMarco. Why Does Software Cost So Much? Dorset House, 1995, pg. 173-178.

There are two familiar variants of the documentation problem: a) you write all the internal documentation that you know you need and you pay a terrible price for it, or b) you don't write all the internal documentation you need and you pay a terrible price for that.

The first variant occurs in companies that have a strong commitment to the conventional wisdom of software construction. They respect, as an article of faith, the maxim: "The job is not over until the paperwork is done." But the costs of paper documentation of program products can be daunting. ... The costs of not documenting program internals are less well quantified, but most development managers believe that missing and inadequate internal documentation are major contributors to the staggering cost of software maintenance.

In early 1988, Aldus Engineering's Product Adaptation group embarked on an experiment in documenting program internals on video. The perception was that this medium had a lower drudgery factor for those who had to produce the documentation, and a lower total cost. Proof of the first of these points was the enthusiasm the idea provoked among those who were asked to participate. ... The initial videos were planned as five afternoon filming sessions, covering data structures, display issues, screen dynamics, event handling, and portability concerns. ... In an informal assessment, viewers of the videos provided positive feedback.

Penny Grubb and Armstrong Takang. Software Maintenance: Concepts and Practice, 2003, pg 7, 239-243.

To understand software maintenance we need to be clear about what is meant by 'software'. It is a common misconception to believe that software is programs. Software comprises not only programs - source code and object code - but also documentation of any facet of the program such as requirements analysis, specification, design, system and user manuals, and the procedures used to setup and operate the software system.

Table 7.1 Types and functions of documentation

User documentation

1. System overview	Provides general description of system functions
2. Installation guide	Describes how to set up the system, customize it to local needs, and configure it to particular hardware and other software systems
3. Beginner's guide / tutorial	Provides simple explanations of how to start using the system
4. Reference guide	Provides in-depth description of each system facility and how it can be used
5. Enhancement	

booklet / release notes	Contains a summary of new features
6. Quick reference card	Serves as a factual lookup
7. System administration	Provides information on services such as networking, security, and upgrading

System documentation

1. System rationale	Describes the objectives of the entire system
2. Requirements analysis / specification	Provides information on the exact requirements for the system as agreed between the stakeholders (user, client, customer, developer).
3. Specification / design	Provides description: (i) of how the system requirements are implemented (ii) of how the system is decomposed into a set of interacting program units (iii) the function of each program unit
4. Implementation	Provide description of: (i) how the detailed system is expressed in some formal programming language (ii) program actions in the form of intra-program comments
5. System test plan	Provides a description of how program units are tested individually and how the whole system is tested after integration
6. Acceptance test plan	Describes the tests the system must pass before the users accept it
7. Data dictionaries	Contains descriptions of all terms that relate to the software system in question

There are a number of reasons why it is of paramount importance to have accurate and sufficient documentation about a software system:

- **To facilitate program comprehension:** The function of documentation in the understanding of subsequent modification of the software cannot be overemphasized. Prior to undertaking any software maintenance work, the maintainer should have access to as much information as possible about the whole system.
- **To act as a guide to the user:** Documentation aimed at users of a system is usually the first contact they have with the system.
- **To complement the system** Documentation forms an integral part of the entire software system. Without the documentation, there is little assurance that the software satisfies stated requirements or that the organization will be able to maintain it.

The cost of maintaining a software system is proportional to the effectiveness of the documentation which describes what the system does as well as the logic used

to accomplish its tasks. To ensure that documentation is up to date, procedures should be put in place to encourage the use of adequate resource for the updating of documents concurrent with system updates.

Mira Kajko-Mattsson. "The State of Documentation Practice within Corrective Maintenance". Proceedings of the IEEE International Conference on Software Maintenance, ICSM 2001, pg. 354-363.

Consistent, correct and complete documentation is an important vehicle for the maintainer to gain understanding of a software system, to ease the learning and/or relearning processes, and to make the system more maintainable. ... For this purpose, we have defined a set of documentation requirements. These requirements have been satisfactorily implemented to the following extent: 53% of the organizations studied deliver complete and consistent system documentation to the maintenance phase. Only 16% of the organizations studied update their software system documents at all granularity levels. 53% of the organizations studied have their user manuals consistent with the state of their software systems. 42% of the organizations studied revise and modify their regression test case repositories. 11% of the organizations studied have achieved full traceability amongst their system documents, and only 5% have achieved full traceability of change. 21% of the organizations studied provide guidelines for how to write system documentation. 42% of the organizations studied define and follow guidelines for internal software code documentation. 26% of the organizations studied provide a checklist of all document types required for executing and supporting the corrective maintenance process. In only 21% of the organizations studied, the corrective maintenance process explicitly states where in the process each type of documentation should be updated/checked. 26% of the organizations studied have a formal approval of the quality of system documentation after each corrective change. 37% of the organizations studied periodically check the quality of their system documentation. 32% of the organizations update their documentation as soon as some inconsistency is discovered. 74% of the organizations studied have access to system development documentation, but only 16% of the organizations have access to system development journal. 68% of the organizations studied record all changes to their software systems. When studying this requirement, we have observed that there are still organizations communicating software problems and changes orally. 32% of the organizations studied keep history of the reported software problems for their software components. 21% of the organizations studied actively update a formal system maintenance journal. None of the organizations studied gives some form of education in written proficiency in order to improve the quality of system documentation.

Timothy Lethbridge, Janice Singer, Andrew Forward. "How Software Engineers Use Documentation: The State of the Practice". IEEE Software, November 2003, pg 35-39.

Percentage of survey respondents who rated documentation effective or extremely effective for particular tasks: learning a software system (61%), testing a software system (58%), working with a new software system (54%), solving problems when other developers are unable to answer questions (50%), looking for big-picture information about a software system (46%), maintaining a software system (35%), answering questions about a system for management or customers (33%), looking for in-depth information about a software system (32%), and working with an established software system (32%).

Our results indicate that software engineers create and use simple yet powerful documentation, and tend to ignore complex and time-consuming documentation.

Harvey Siy and Lawernce Votta. "Does the Modern Code Inspection Have Value?". Proceedings of the IEEE International Conference on Software Maintenance, ICSM 2001, pg. 281.

For years, it was believed that the value of inspections is in finding and fixing defects early in the development process. Otherwise, the cost to find and fix them later is much higher. However, in examining code inspection data, we are finding that inspections are beneficial for an additional reason. They make code easier to understand and change. An analysis of data from recent code inspection experiments shows that 60% of all issues raised in the code inspections are not problems that could have been uncovered by latter phases of testing or field usage because they have little or nothing to do with the visible execution behavior of the software. Rather, they improve the maintainability of the code by making the code conform to coding standards, minimizing redundancies, improving language proficiency, improving safety and portability, and raising the quality of documentation. We conclude that even if advances in software technology have diminished the value of inspections as a defect detection tool, in most cases, it continues to be of value as a maintenance tool.

Agile Documentation

Ron Jeffries. Essential XP: Documentation

Extreme Programming (XP) has very high focus on incremental development. The development cycle is to begin with Simple Design, communicated through Pair Programming, supported by extensive Unit Tests, and evolved through Refactoring. For this to work, it must be possible to refactor the code: the code must be very clean and very clear. There are rules (see elsewhere a discussion of the rules of simplicity relating to reusability) that help programmers produce clear code. In essence, if there is an idea in our head when we write the code, we require ourselves to express that idea directly in the code. We treat the need for comments as "code smells" suggesting that the code needs improvement, because it isn't clear enough to stand alone. Our first action is to improve the code to be more clear. Whatever we cannot make clear in code, we then comment.

Unit tests are also documentation. The unit tests show how to create the objects, how to exercise the objects, and what the objects will do. This documentation, like the acceptance tests belonging to the customer, has the advantage that it is executable. The tests don't say what we think the code does: they show what the code actually does.

Scott Ambler. "All the Right Questions". Software Development, December 2002, pg. 44.

Extreme programming doesn't prevent you from writing documentation-it insists only that you justify why you need it. Documentation should be treated like any other requirement-estimated, prioritized, and planned for accordingly. In other words, don't blindly create documentation simply because it makes you feel comfortable; instead do it because it adds value. Visit agile documentation for more information.

Andreas Ruping. Agile Documentation, John Wiley, 2003, pg. 197-204.

- **Focused Information:** A clear and identifiable focus on a particular topic makes a document concise and straightforward.
- **Individual Documentation Requirements:** The most effective approach towards documentation is for each project to define its documentation requirements individually.
- **Focus on Long-Term Relevance:** There is much value in documentation that focuses on issues with a long-term relevance.
- **Design Rationale:** Design documents become a valuable source of information if they aren't restricted to describing the actual design, but also focus on the rationale behind the design and explain why the particular design was chosen.
- **The Big Picture:** A good feel for a project is best conveyed through a description of the 'big picture' of the architecture that underlies the system under construction.
- **Separation of Description and Evaluation:** Authors gain credibility if, in their documents, they clearly separate description from evaluation.
- **Realistic Examples:** Project documents are much more convincing if they include realistic examples from the project's context.
- **Structured Information:** Most project documents are best organized as sequential yet well-structured text. This begins with well-chosen chapters and sections, but may well extend to using textual building blocks consistently throughout a document.
- **Judicious Diagrams:** Diagrams can provide excellent overviews, while an accompanying text explains details to the extent that is necessary.
- **Code-Comment Proximity:** Documentation of the code, to the extent that a project team considers it necessary, is best done through source code comments. Separate documents should be reserved for higher-level issues such as overviews, requirements, design, and architecture.

- **A Distinct Activity:** When documentation is considered a distinct project activity, and not just the by-product of coding, it can be assigned its own budget, priority, and schedule. Documentation can then be weighed against other project activities.
- **Writing and Reflection:** To get the best out of documentation, team members have to spend time on the actual writing, as well as in reflection on what they have written, preferably in an undisturbed environment.
- **Review Culture:** Documentation can profit from reviews, provided a review culture has been established in which both authors and reviewers feel comfortable.
- **Information Marketplace:** Documents gain more attention if the intended readers are actively invited to read them.

Kent Beck. "A Theory of Programming". Dr. Dobb's Journal, November 2007

Code communicates well when a reader can understand it, modify it, or use it. While programming it's tempting to think only of the computer. However, good things happen when I think of others while I program. I get cleaner code that is easier to read, it is more cost-effective, my thinking is clearer, I give myself a fresh perspective, my stress level drops, and I meet some of my social needs. Part of what drew me to programming in the first place was the opportunity to commune with something outside myself. However, I didn't want to deal with sticky, inexplicable, annoying human beings. Programming as if people didn't really exist paled after only a couple of decades. Building ever-more-elaborate sugar castles in my mind became colorless and stale.

One of the early experiences that led me to focus on communication was discovering Knuth's Literate Programming: a progam should read like a book. It should have plot, rhythm, and delightful little turns of phrase. When Ward Cunningham and I first read about literate programs, we decided to try it. We sat down with one of the cleanest pieces of code in the Smalltalk image, the ScrollController, and tried to make it into a story. Hours later we had completely rewritten the code on our way to a reasonable paper. Every time a bit of logic was a little hard to explain, it was easier to rewrite the code than explain why the code was hard to understand. The demands of communication changed our perspective on coding.

There is a sound economic basis for focusing on communication while programming. The majority of the cost of software is incurred after the software has been first deployed. Thinking about my experience of modifying code, I see that I spend much more time reading the existing code than I do writing new code. If I want to make my code cheap, therefore, I should make it easy to read.

Scott Ambler. "What's Normal? Models to Documentation", Software Development's Agile Modeling Newsletter, December 2004.

In Agile Modeling (AM), you want to travel as light as possible, so you need to choose the best artifact to record information. (I use the term "artifact" to refer to any model, document, source code, plan or other item created during a software development project.) Furthermore, you want to record information as few times as possible; ideally only once. For example, if you describe a business rule in a use case, then describe it in detail in a business rule specification, then implement it in code, you have three versions of the same business rule to maintain. It would be far better to record the business rule once, ideally as human-readable but implementable code, and then reference it from any other artifact as appropriate.

Why do you want to record a concept once? First, the more representations that you maintain, the greater the maintenance burden, because you'll want to keep each version in sync. Second, you'll have greater traceability needs with multiple copies because you'll have to relate each version to its alternate representations to keep them synchronized when a change does occur. AM does advise you to update only when it hurts, but having multiple copies of something means you?re likely to feel the pain earlier and more often. Finally, the more copies you have, the greater the chance you'll end up with inconsistent information, as you probably won't be able to keep the versions synchronized.

It's interesting that traditional processes typically promote multiple recordings of technical information, such as representing business rules three different ways, while also prescribing design concepts such as normalization and cohesion that promote developing designs that implement concepts once. For example, the rules of data normalization motivate you to store data in one place and one place only. Similarly, in object-oriented and component-based design, you want to build cohesive items (components, classes, methods and so on) that fulfill only one goal. If this is O.K. for your system design, shouldn?t it also be O.K. for your software development artifacts? We clearly need to rethink our approach.

It's clear that you should store system information in one place and one place only, ideally in the most effective place. In software development, this concept is called "artifact normalization"; in the technical documentation world, it's called "single sourcing". With single sourcing, you aim to record technical information only once and then generate various documentation views as needed from that information. A business rule, for example, would be recorded using some sort of business-rule definition language. A human-readable view would be generated for your requirements documentation (easier said than done, but for the sake of argument, let's assume it's possible) and an implementation view generated that would be run by your business rule engine or compiled as application source code.

To make the single-sourcing vision work, you need a common way to record information. Darwin Information Typing Architecture (DITA) is an XML-based format that's promoted for single-sourced technical documentation. Nothing can stop you from creating your own storage strategy: Single sourcing is often approached in a top-down manner with the data structure for the documentation typically defined early in a project. However, you can take a more agile approach in which the structure emerges over time.

The primary challenge with traditional single sourcing? It requires a fairly sophisticated approach to technical documentation. This is perfectly fine, but unfortunately, many organizations aren?t yet able to achieve this vision and eventually back away from the approach. This doesn?t mean that you need to throw out the baby with the bath water; you should still strive to normalize all of your software development artifacts.

Just as it's extremely rare to find a perfectly normalized relational database, I suspect that you'll never truly be able to fully normalize all of your software artifacts. In the case of databases, performance considerations (and to be fair, design mistakes made by project teams) result in less-than-normal schemas. Similarly, everyone won't be able to work with all types of artifacts -- it isn?t realistic to expect business stakeholders to be able to read program source code and the über-tools required to support this vision continue to elude us.

Charles Connell. It's Not About Lines of Code.

Everyone wants programmers to be productive. Managers of programmers want maximum productivity -- it gets the work done faster and makes the managers look good. All other things being equal, programmers like being productive. They can get home earlier, reduce stress during the workday, and feel better about their finished products.

Lines of code per day -- This is the classic definition of software productivity for individual programmers. Unfortunately, as other authors have noted as well, the definition makes little sense. Assume Fred's code is of such poor quality that, for each day of work he does, someone else must spend five days debugging the code. Is Fred highly productive?
Lines of correct code per day -- This definition adjusts for the problem of a programmer producing lousy code. Suppose Fred cleans up his act. Imagine his code is completely bug-free, but contains no comments at all. The next programmer to take over Fred's code will find it impenetrable, and possibly will be forced to rewrite the code in order to add any new features. Is Fred productive now?
Lines of correct, well-documented code per day -- This definition gets closer to what we want, but something still is missing. Imagine both Fred and another programmer, Danny Designer, are given similar assignments. Danny also completes his assignment in five days, but he writes only 500 lines of code (all correct and well-documented). Who was more productive? Probably Danny. His code is shorter and simpler, and simplicity is almost always better in engineering.
Lines of clean, simple, correct, well-documented code per day -- This is a pretty good definition of productivity, and one many experienced, savvy technical managers would accept. There is still something about this definition though that misses what software engineers ultimately are trying to do. Imagine Fred, Danny, and a third programmer, Ingrid Insightful, are given similar assignments. Something about the assignment bothers Ingrid however, so she decides to go outside for a walk. Ingrid wakes up with a start and realizes what bothers her about the programming assignment: this new feature is suspiciously like an

existing feature. Ingrid opens up the source code for the existing feature and begins deleting large sections of it. Before long, she has generalized the existing feature so it is simpler, more intuitive, and includes the new capabilities she was asked to add. In the process, she has reduced the code base by 2000 lines. Who was more productive on this day? Certainly Ingrid. Getting away from a problem sometimes is a good way to solve it. And programmers who understand the big picture make smarter decisions, because they are able to reuse code and combine features effectively.

Ability to solve customer problems quickly -- This is the true definition of programmer productivity, and is what Ingrid accomplished in the example. What software engineering really is about is solving problems for the people who will use the software. Any other definition of programmer productivity misses the mark. This definition raises a difficult question though: If a programmer can be highly productive by writing a negative amount of code, how do we measure productivity for software engineers? There is no easy answer to this question, but the answer surely is not a rigid formula related to lines of code, bug counts, or face time in the office. Each of these measures has some value for some purposes, but managers should not lose site of what software engineers are doing. They are creating machines to solve human problems.

Design Documentation

Bjarne Stroustrup. The C++ Programming Language, Addison Wesley, 2000, pg. 694.

The purpose of professional programming is to deliver a product that satisfies its users. The primary means of doing so is to produce software with a clean internal structure and to grow a group of designers and programmers skilled enough and motivated enough to respond quickly and effectively to change and opportunities.

Why? The internal structure of the program and the process by which it was created are ideally of no concern to the end user. Stronger: if the end user has to worry about how the program was written, then there is something wrong with the program. Given that, what is the importance of the structure of a program and of the people who create the program? A program needs a clean internal structure to ease: testing, porting, maintenance, extension, reorganization, and understanding.

The main point is that every successful piece of software has an extended life in which it is worked on by a succession of programmers and designers, ported to new hardware, adapted to unanticipated uses, and repeatedly reorganized. Throughout the software's life, new versions of it must be produced with acceptable error rates and on time.

David Parnas and Paul Clements. "A Rational Design Process: How and Why to Fake It" in Software State-of-the-Art: Selected Papers. Dorset House, 1990, pg. 353-355.

It should be clear that documentation plays a major role in the design process that we are describing. Most programmers regard documentation as necessary evil, written as an afterthought only because some bureaucrat requires it. They do not expect it to be useful.

This is a self-fulfilling prophesy; documentation that has not been used before it is published, documentation that is not important to its author, will always be poor documentation.

Most of that documentation is incomplete and inaccurate, but those are not the main problems. If those were the main problems, the documents could be easily corrected by adding or correcting information. In fact, there are underlying organizational problems that lead to incompleteness and incorrectness and those problems, which are listed below, are not easily repaired.

1) Poor Organization: Most documentation today can be characterized as "stream of consciousness" and "stream of execution." "Stream of consciousness" writing puts information at the point in the text that the author was writing when the thought occurred to him. "Stream of execution" writing describes the system in the order things will happen when it runs. The problem with both of these documentation styles is that subsequent readers cannot find the information they seek. It will therefore not be easy to determine that facts are missing, or to correct them when they are wrong. It will not be easy to find all the parts of the document that should be changed when the software is changed. The documentation will be expensive to maintain and, in most cases, will not be maintained.

2) Boring Prose: Lots of words are used to say what could be said by a single programming language statement, a formula, or a diagram. Certain facts are repeated in many different sections. This increases the cost of the documentation and its maintenance. More importantly it leads to inattentive reading and undiscovered errors.

3) Confusing and Inconsistent Terminology: Any complex system requires the invention and definition of new terminology. Without it the documentation would be far too long. However, the writers of software documentation often fail to provide precise definitions for the terms they use. As a result, there are many terms used for the same concept and many similar but distinct concepts described with the same term.

4) Myopia: Documentation that is written when the project is nearing completion is written by people ho have lived with the system for so long that they take major decisions for granted. They document the small details that they think they will forget. Unfortunately, the result is a document useful to people who know the system well, but impenetrable for newcomers.

Documentation in the ideal design process meets the needs of the initial developers as well as the needs of the programmers who come later. Each of the documents mentioned above records requirements or design decisions and is used as a reference document for the rest of the design. However, they also provide the

information that the maintainers will need. Because the documents are used as reference manuals throughout the building of the software, they will be mature and ready to use in later work. The documentation in this design process is not an afterthought; it is viewed as one of the primary products of the project. Some systematic checks can be applied to increase completeness and consistency. [...]

"Stream of consciousness" and "stream of execution" documentation is avoided by designing the structure of each document. Each document is designed by stating the questions that it must answer and refining the questions until each defines the content of an individual section. There must be one, and only one, place for every fact that will be in the document. The questions are answered, i.e., the document is written, only after the structure of the document has been defined. When there are several documents of a certain kind, a standard organization is written for those documents. Every document is designed in accordance with the same principle that guides our software design: separation of concerns. Each aspect of the system is described in exactly one section and nothing else is described in that section. When documents are reviewed, they are reviewed for adherence to the documentation rules as well as for accuracy.

The resulting documentation is not easy or relaxing reading, but it is not boring. It makes use of tables, formulas, and other formal notation to increase the density of information. The organizational rules prevent the duplication of information. The result is documentation that must be read very attentively, but rewards its reader with detailed and precise information. [...]

No matter how often we stumble on our way, the final documentation will be rational and accurate. Even mathematics, the discipline that many of us regard as the most rational of all follows this procedure. [...] Analogous reasoning applies to software. Those who read the software documentation want to understand the programs, not relive their discovery. By presenting rationalized documentation we provide what they need.

Our documentation differs from the ideal documentation in one important way. We make a policy of recording all of the design alternatives that we considered and rejected. For each, we explain why it was considered and why it was finally rejected. Months, weeks, or even hours later, when we wonder why we did what we did, we can find out. Years from now, the maintainer will have many of the same questions and will find his answers in our documents.

David Parnas. "Why Software Jewels Are Rare". Computer, February 1996, pg. 57-60.

For much of my life, I have been a software voyeur, peeking furtively at other people's dirty code. Occasionally, I find a real jewel, a well-structured program written in a consistent style, free of kludges, developed so that each component is simple and organized, and designed so that the product is easy to change. Why, since we have been studying software construction for more than 30 years, don't we find more such jewels? How often is it possible to produce such a jewel of a

system? Seldom? Frequently? Always?

Often, software has grown large and its structure degraded because designers have repeatedly modified it to integrate new systems or add new features. ... Wirth suggests that we keep our software lean by sticking to essentials, omitting "bells and whistles". Besides, lean software is likely to be smaller and even faster.

However, it isn't always necessary to choose between function and elegance. Perhaps, I'm too optimistic, but I don't think a designer must omit features to build what Wirth calls "lean software". What is necessary is to design the product so that newly added features do not eliminate useful capabilities, make good use of capabilities already present for other purposes, and can be ignored or deleted by people who don't want them. ... Given a choice between tool and jewel, we will choose tool; but with a little more thought, we can often have both. Studying the jewels can show us how.

One of the weaknesses of technological society is that we sometimes place far too much emphasis on originality. Creativity and originality are obviously valuable wherever there is room for improvement, and they are essential when dealing with problems for which we have no adequate solution.

Nevertheless, we have an unfortunate tendency to value creativity as an end in itself and use it as an excuse for ignorance. I have known both researchers and developers who refused to look at previous work because they wanted to use their own ideas. Managers often do not allow their designers time to study the way things have been done in the past. It seems obvious that we should use our own ideas only if they are better than previous ones. Successful innovators usually know previous work and have managed to understand the fundamental weakness in earlier approaches. Too many software products show evidence of "ignorant originality." They make the same mistakes others made before them and ignore solutions that others have found.

Sometimes new languages are used in the design of jewels, and authors may attribute a product's success to the use of a particular language or type of language. Here, I have grave doubts. ... My experience does not support the view that the programming language used determines the quality of the software. I have seen beautiful, lean software written only using assembler (Dijkstra offers an example), good software written in Fortran, and even good software written in C. I have also seen programs in which each of these tools was used badly. ... Focusing on the programming language is a red herring that will distract us from real solutions to the problem of poor software. The jewels I've found owe their elegance to: the use of good decomposition principles, the use of good hierarchical structures, and the design of interfaces.

There is much to learn from jewel-like systems. ... The most important lesson is "up-front investment." In each of the jewels I've seen, the designers had obviously spent a lot of time thinking about the structure of the system before writing code. The system structure could be accurately described and documented without reference to the code. Programs were not just written; they had been planned,

often in some pseudocode or a language other than the actual programming language. ... My engineering teachers laid down some basic rules:

- Design before implementing.
- Document your design.
- Review and analyze the documented design.
- Review implementation for consistency with the design.

These rules apply to software at least as much as they do to circuits or machines.

Paul Clements, Felix Bachmann, Len Bass, David Garlan, James Ivers, Reed Little, Robert Nord, and Judith Stafford. Documenting Software Architectures: Views and Beyond, Addison-Wesley, 2003, pg 9, 13, 24.

Documentation connotes the creation of an artifact: namely, a document, which may, of course, be electronic files, Web pages, or paper. Thus, documenting a software architecture becomes a concrete task: producing a software architecture document. Viewing the activity as creating a tangible product has advantages. We can describe good architecture documents and bad ones. We can use completeness criteria to judge how much work is left in producing this artifact and determining when the task is done. Planning or tracking a project's progress around the creation of artifacts, or documents, is an excellent way to manage. Making the architecture information available to its consumers and keeping it up-to-date reduces to a solved problem of configuration control. Documentation can be formal or not, as appropriate. Documents may describe, or they may specify. Hence, the term is appropriately general.

Finally, documentation is a longstanding software engineering tradition. Documentation is the task that you are supposed to do because it's good for you. It's what your software engineering teachers taught you to do, your customers contracted you to do, your managers nagged you to do, and what you always found a way not to do. So if documentation brings up too many pangs of professional guilt, use any term you like that's more palatable. The essence of the activity is writing down--and keeping current--the results of architectural decisions so that the stakeholders of the architecture--people who need to know what it is to do their job--have the information in accessible, nonambiguous form.

Perhaps the most important concept associated with software architecture documentation is that of the view. A software architecture is a complex entity that cannot be described in a single one-dimensional fashion. Our analogy with the bird wing proves illuminating. There is no single rendition of a bird wing. Instead there are many: feathers, skeleton, circulation, muscular views, and many others. Which one of these views is the "architecture" of the wing? None of them. Which views convey the architecture? All of them.

We use the concept of views to give us the most fundamental principle of architecture documentation: Documenting an architecture is a matter of

documenting the relevant views and then adding documentation that applies to more than one view.

What are relevant views? It depends on your goals. As we saw previously, architecture documentation can serve many purposes: a mission statement for implementers, a basis for analysis, the specification for automatic code generation, the starting point for system understanding and asset recovery, or the blueprint for project planning. The views you document depend on the uses you expect to make of the documentation.

It may be disconcerting that no single view can fully represent an architecture. Additionally, it feels somehow inadequate to see the system only through discrete, multiple views, that may or may not relate to one another in a straightforward way. The essence of architecture is the suppression of information not necessary to the task at hand, and so it is somehow fitting that the very nature of architecture is such that it never presents its whole self to us but only a facet or two at a time. This is its strength: Each view emphasizes certain aspects of the system while deemphasizing or ignoring other aspects, all in the interest of making the problem at hand tractable. Nevertheless, no one of these individual views adequately documents the software architecture for the system. That is accomplished by the complete set of views along with information that transcends them.

Seven rules for sound documentation: 1) Write documentation from the reader's point of view. 2) Avoid unnecessary repetition. 3) Avoid ambiguity. 4) Use a standard organization. 5) Record rationale. 6) Keep documentation current but not too current. 7) Review documentation for fitness of purpose.

Paul Dilascia. "What Makes Good Code Good?". MSDN Magazine, July 2004, pg. 144.

A good program works flawlessly and has no bugs. But what internal qualities produce such perfection? It's no mystery, we just need some occasional reminding. Whether you code in C/C++, C#, Java, Basic, Perl, COBOL, ASM, all good programming exhibits the same time-honored qualities: simplicity, readability, modularity, layering, design, efficiency, elegance, and clarity.

Simplicity means you don't do in ten lines what you can do in five. It means you make extra effort to be concise, but not to the point of obfuscation. It means you abhor open coding and functions that span pages. Simplicity--of organization, implementation, design--makes your code more reliable and bug free. There's less to go wrong.

Readability means what it says: that others can read your code. Readability means you bother to write comments, to follow conventions, and pause to name your variables wisely.

Modularity means your program is built like the universe. The world is made of molecules, which are made of atoms, electrons, nucleons, and quarks. Likewise

good programs erect large systems from smaller ones, which are built from even smaller building blocks. And just as atoms combine in novel ways, software components should be reusable.

Layering means that internally, your program resembles a layer cake. The app sits on the framework sits on the OS sits on the hardware. Even within your app, you need layers, like file-document-view-frame. Higher layers call ones below, which raise events back up. (Calls go down; events go up.) Lower layers should never know what higher ones are up to. The essence of an event/callback is to provide blind upward notification. If your doc calls the frame directly, something stinks. Modules and layers are defined by APIs, which delineate their boundaries. Thus, design is critical.

Design means you take the time to plan your program before you build it. Thoughts are cheaper than debugging. A good rule of thumb is to spend half your time on design. You need a functional spec (what the program does) and an internal blueprint. APIs should be codified in writing.

Efficiency means your program is fast and economical. It doesn't hog files, data connections, or anything else. It does what it should, but no more. It loads and departs without fuss. At the function level, you can always optimize later, during testing. But at high levels, you must plan for performance. If the design requires a million trips to the server, expect a dog.

Elegance is like beauty: hard to describe but easy to recognize. Elegance combines simplicity, efficiency, and brilliance, and produces a feeling of pride. Elegance is when you replace a procedure with a table, or realize that you can use recursion--which is almost always elegant.

Clarity is the granddaddy of good programming, the platinum quality all the others serve. Computers make it possible to create systems that are vastly more complex than physical machines. The fundamental challenge of programming is managing complexity. Simplicity, readability, modularity, layering, design, efficiency, and elegance are all time-honored ways to achieve clarity, which is the antidote to complexity.

Clarity of code. Clarity of design. Clarity of purpose. You must understand--really understand--what you're doing at every level. Otherwise you're lost. Bad programs are less often a failure of coding skill than of having a clear goal. That's why design is key. It keeps you honest. If you can't write it down, if you can't explain it to others, you don't really know what you're doing.

Source Code Comments

Jeffrey Kotula. Source Code Documentation: An Engineering Deliverable" in Proceedings of the Technology of Object-Oriented Languages and Systems, 2000.

Source code documentation is a fundamental engineering practice critical to efficient software development. Regardless of the intent of its author, all source code is eventually reused, either directly, or just through the basic need to understand it. In either case, the source code documentation acts as a specification of behavior for other engineers. Without documentation, they are forced to get the information they need by making dangerous assumptions, scrutinizing the implementation, or interrogating the author. These alternatives are unacceptable. Although some developers believe that source code "self-documents", there is a great deal of information about code behavior that simply cannot be expressed in source code, but requires the power and flexibility of natural language to state. Consequently, source code documentation is an irreplaceable necessity, as well as an important discipline to increase development efficiency and quality.

Richard Gunderman. "A Glimpse into Program Maintenance" in Techniques of Program and System Maintenance. QED Information Sciences, 1988, pg. 59.

Program documentation has been propelled into importance by sheer necessity. However, it still suffers from glowing tributes but inept implementations. One of the basic elements of good program documentation is an effective program listing.

Edward Yourdon. "Flashes on Maintenance From Techniques of Program Structure and Design" in Techniques of Program and System Maintenance. QED Information Sciences, 1988, pg. 73.

In my opinion, there is nothing in the programming field more despicable than an uncommented program. A programmer can be forgiven many sins and flights of fancy, including those listed in the sections below; however no programmer, no matter how wise, no matter how experienced, no matter how hard-pressed for time, no matter how well-intentioned, should be forgiven an uncommented and undocumented program.

Of course, it is important to point out that comments are not an end unto themselves. As Kernighan and Plauger point out in their excellent book, The Elements of Program Style, good comments cannot substitute for bad code. However, it is not clear that good code can substitute for comments. That is, I do not agree that it is unnecessary for comments to accompany "good" code. The code obviously tells us what the program is doing, but the comments are often necessary for us to understand why the programmer has used those particular instructions.

Robert Dunn. Software Defect Removal. McGraw-Hill, 1984, pg. 308.

Common sense also leads us to the recognition of the characteristics of programs that makes the programs maintainable. Above all, we look for programs that

exhibit logical simplicity -- failing that, at least clarity. The earmarks of simplicity and clarity include modularity (true functional modularity, not arbitrary segmentation) and a hierarchical control structure, restrictions on each module's access to data, structured data forms, the use of structured control forms, and generous and accurate annotation.

Much has been said of the technical members of this set in earlier pages. Of good annotation, there are several features that must be included. First, the header information of each procedure should provide a concise statement of the procedure's external specifications, including a description of input and output data. Each section of the procedure should be introduced by comments identifying the section's relation to the external characteristics. Finally, comments within each section should relate groups of statements to the program's documented description. This last is automatically achieved by using design language statements as source code comments.

David Zokaities. "Writing Understandable Code". Software Development, January 2002, pg. 48-49.

Software must be understandable to two different types of entities for two different purposes. First, compilers or interpreters must be able to translate source code into machine instructions. Second, people need to understand the software so they can further develop, maintain and utilize the application. The average developer overemphasizes capability and function while undervaluing the human understanding that effects improved development and continued utilization. There should be a description in clear view within the programming medium.

As I gradually improved my in-code documentation, I realized that English is a natural language, but computer languages, regardless of how well we use them, are still "code." Communication via natural language is a relatively quick and efficient process. Not so with computer languages: They must be "decoded" for efficient human understanding.

People who read my code? wait a moment - did I say "read my code?" Now that's a remarkable way to approach software - not to debug, analyze, program, or develop, but simply to read. The act of reading allows me to approach my code as a work of software art: I strive to make the overall design, algorithm, structure, documentation and style as simple, elegant, through and effective as practical. Yes, this takes time, but when I'm rushed, I usually dash off the wrong implementation of the wrong design, and the darn project takes twice as long as it would have had I done it right in the first place. A disciplined, focused approach clarifies my thinking and improves my implementation. In keeping with a reasonable attempt for excellence, I proofread my applications.

My goal is to find a balance, describing all salient program features comprehensively but concisely. I explained each software component's purpose, the algorithm used, arguments, inputs, outputs- even the reason for #include-ing a particular header file. I document each section of each function so that the overall

program flow is readily understandable.

David Zokaities. "Feedback". Software Development, March 2002, pg. 14.

My article seems to have generated quite a bit of controversy. The article's text received only praise. This implies that the goal of "understandable code" is well-nigh universal. How best to achieve it seems to be a matter of highly polarized opinion. Even for the wealth of comments that I customarily provide, some readers chided me for not having enough! Other readers believe that all comments are superfluous and cause trouble by their very existence. I've seen horrendous maintenance problems incurred with this approach. Some readers believe that Design by Contract, coupled with lengthy function and variable names, provides all the necessary documentation.

My experience is that well-developed modular designs, coupled with good system documentation, descriptive identifier names and a natural-language narrative, result in code that's a pleasure to work with and efficient to maintain.

Henry Ledgard. Professional Software Volume II: Programming Practice, Addison-Wesley, 1987, pg 65.

A program is in some sense a permanent object in that it can have a long lifetime. For the future reader, comments in a program should be truly substantive. They should say something. They should assist the program reader.

The professional thinks of a comment as a way to proceed from one point (a given state of knowledge) to another (understanding what is written in the program). The comment is a bridge. The professional assumes something about the reader of the program--the reader being, of course, someone else. It is fair to assume that the reader knows the language in which the program is written. The reader's difficulty is to modify the program at hand.

These observations lead to some specific recommendations. First, regarding the idea of comments as a bridge--Extensive introductory program comments are entirely in order. These comments set the stage for reading the program. They may contain an outline of the solution adopted by the programmer, summarize its input and output, give a directory of key variable names, or describe an algorithm that may not be known to the reader. Such comments provide a direct bridge from the problem to the program. They do not intrude on the reading of the program itself because they appear at the beginning of the program and can be read or not as the reader desires.

A second recommendation has to do with procedures and other major units of the program--Introductory module comments are also in order. Comments following a procedure header explaining the general nature of the procedure are not only in order but may be necessary. Keeping in mind the bridge aspect, we need not describe the calling environment. The professional assumes that the reader has

read the program to the degree that the procedure calls are understood--but maybe not the procedure itself. As such, the procedure header comments should be short and help the reader understand the next level of detail in the program.

Third, the professional should spend the most energy on the code itself. This means--Avoid embedded (in-line) comments within the body of the module itself. It is my view that such comments can readily intrude upon the meaning of a program. Ideally, the code should speak for itself and require few supporting comments.

Dennis Smith. Designing Maintainable Software. Springer-Verlag, 1999, pg. 103.

Documentation that is structured and contained within the program is able to immediately satisfy the changing information demands of the maintainer. Those needs are determined in part by the subtask on which he is currently working. For solving nontrivial error correction or modification problems, the maintainer must have a detailed understanding of the program. To locate a section of code, knowledge of the program's structure is required. Knowing how an instruction sequence relates to other parts of the program is important for altering and testing software. The documentor can inform the unknowledgeable programmer in each subtask demand by varying the message content and effectively using the visual space.

Information may be conveyed to the maintainer in several ways. One is an abstract summary of the module at the beginning of the routine. Another is through the titles and headings of processing sections positioned in the instruction sequence. The third is in phrases and short sentences to the right of the code. They describe the processing steps and relate them to other parts of the program. The descriptions are organized into an outline that reflects the processing divisions of the routine.

The size and complexity of the module determine whether the information will be used. Small routines may need only comments to the right of the code. A more complete description is required for large programs.

The type of documentation that has just been read has a bearing on the processing of code. Documentation formats act as advance organizers of thought. Each type primes the maintainer for a different response to the instructions encountered. Messages that are consistent with the structure of the program aid recognition and recall. ... Programs are documented to enhance the maintainer's performance.

Penny Grubb and Armstrong Takang. Software Maintenance: Concepts and Practice, 2003, pg 7, 120-121.

Program comments within and between modules and procedures usually convey information about the program, such as the functionality, design decisions, assumptions, declarations, algorithms, nature of input and output data, and

reminder notes. Considering that the program source code may be the only way of obtaining information about a program, it is important that the programmers should accurately record useful information about these facets of the program and update them as the system changes. Common types of comments used are prologue comments and in-line comments. Prologue comments precede a program or module and describe goals. In-line comments, within the program code, describe how these goals are achieved.

The comments provide information that the understander can use to build a mental representation of the target program. For example, in Brooks' top-down model, comments - which act as beacons - help the programmer not only form hypothesis, but refine them to closer representations of the program. Thus, theoretically there is a strong case for commenting programs. The importance of comments is further strengthened by evidence that the lack of good comments in programs constitutes one of the main problems that programmers encounter when maintaining programs. It has to be pointed out that comments in programs can be useful only if they provide additional information. In other words, it is the quality of the comment that is important, not its presence or absence.

Christopher Seiwald. "Pillars of Pretty Code". Software Development, January 2005, pg 49-51.

The essence of pretty code? One can infer much about its structure from a glance, without completely reading it. I call this visual parsing: discerning the flow and relative importance of code from its shape.

- **Blend In:** Code changes should blend in with the original style.
- **Bookish:** Keep columns narrow.
- **Disentangle Code Blocks:** Break code into logical blocks within functions, and disentangle the purpose of separate blocks, so that each does a single thing or single kind of thing. A reader can avoid a total reading if a cursory inspection can reveal the whole block's nature.
- **Comment Code Blocks:** Set off code blocks with white space and comments that describe each block. Sometimes large code blocks (with multiline comments) may embed small blocks (with single line comments). Comments should rephrase what happens in the code block, rather than be a literal translation into English. That way, even if your code is inscrutable and your comments gibberish, the reader can at least attempt to triangulate on the actual purpose. Big comments are needed for subtle or problematic code blocks, not necessarily big code blocks.
- **Declutter:** Reduce, reduce, reduce. Remove anything that will distract the reader.
- **Make Alike Look Alike:** Two or more pieces of code that do the same or similar thing should be made to look the same. Nothing speeds the reader along better than seeing a pattern.
- **Overcome Indentation:** The left edge of the code defines its structure, while the right side holds the detail. You must fight indentation to safeguard this property. Code that moves too quickly from left to right (and back again)

mixes major control flow with minor detail.

David Marin. "What Motivates Programmers to Comment?"

Though programmers are often encouraged to comment their source code more thoroughly, there has been very little scientific investigation into what kinds of situations actually cause programmers to do so. I conducted a statistical study of the CVS repositories of nine Open Source projects, and made four major findings. First, the rate at which programmers comment varies widely from project to project and programmer to programmer; even the same programmer will comment at different rates on different projects. Second, programmers tend to comment larger modifications to source code more thoroughly. Third, more programmers modifying the same file does not, in general, mean more commenting. Finally, programmers tend to comment more when they are modifying code that is thoroughly commented to begin with. I then determined through an experiment with programmers that there is a causal link behind my last finding; that is, the more throughly a source code file is commented, the more thoroughly programmers will comment when they make modifications to it.

Randall Hyde. Write Great Code: Understanding the Machine, No Starch Press, 2004, pg. 6.

Here are some attributes of great code:

Uses the CPU efficiently (which means the code is fast)
Uses memory efficiently (which means the code is small)
Uses system resources efficiently
Is easy to read and maintain
Follows a consistent set of style guidelines
Uses an explicit design that follows software engineering conventions
Is easy to enhance
Is well-tested and robust (meaning that it works)
Is well-documented

Software Aging

Frederick Brooks. The Mythical Man-Month. Addison-Wesley, 1995, pg. 169.

A basic principle of data processing teaches the folly of trying to maintain independent files in synchronism. It is far better to combine them into one file with each record containing all the information both files held concerning a given key.

Yet our practice in programming documentation violates our own teaching. We typically attempt to maintain a machine-readable form of a program and an independent set of human-readable documents, consisting of prose and flow

charts.

The results in fact confirm our teachings about the folly of separate files. Program documentation is notoriously poor, and its maintenance is worse. Changes made in the program do not promptly, accurately, and invariably appear in the paper.

The solution, I think, is to merge the files, to incorporate the documentation in the source program. This is at once a powerful incentive toward proper maintenance, and an insurance that the documentation will always be handy to the program user. Such programs are called *self-documenting*.

David Parnas. "Software Aging" in Software Fundamentals. Addison-Wesley, 2001, pg. 557-558, 563.

29.7.1 Keeping Records--Documentation. Even when the code is designed so that changes can be carried out efficiently, the design principles and design decisions are often not recorded in a form that is useful to future maintainers. Documentation is the aspect of software engineering most neglected by both academic researchers and practitioners. It is common to hear a programmer saying that the code is its own documentation; even highly respected language researchers take this position, arguing that if you use their latest language, the structure will be explicit and obvious.

When documentation is written, it is usually poorly organized, incomplete, and imprecise. Often the coverage is random; a programmer or manager decides that a particular idea is clever and writes a memo about it while other topics, equally important, are ignored. In other situations, where documentation is a contractual requirement, a technical writer, who does not understand the system, is hired to write the documentation. The resulting documentation is ignored by maintenance programmers because it is not accurate. Some projects keep two sets of books; there is the official documentation, written as required for the contract, and the real documentation written informally when specific issues arise.

Documentation that seems clear and adequate to its author is often about as clear as mud to the programmer who must maintain the code 6 months or 6 years later. Even when the information is present, the maintenance programmer doesn't know where to look for it. It is almost as common to find the same topic is covered twice, but that the statements in the documentation are inconsistent with each other and the code.

Documentation is not an "attractive" research topic. Last year, I suggested to the leader of an Esprit project who was looking for a topic for a conference, that he focus on documentation. His answer was that it would not be interesting. I objected, saying that there were many interesting aspects to this topic. His response was that the problem was not that the discussions wouldn't be interesting, the topic wouldn't sound interesting and would not attract an audience.

For the past five or six years my own research, and that of many of my students

and close colleagues, has focused on the problems of documentation. We have shown how mathematical methods can be used to provide clear, concise, and systematic documentation of program design. We have invented and illustrated new mathematical notation that is much more suited to use in documentation, but no less formal. The reaction of academics and practitioners to this work has been insight-provoking. Both sides fail to recognize documentation as the subject of our work. Academics keep pointing out that we are neglecting "proof obligations"; industrial reviewers classify our work as "verification" which they (often correctly) consider too difficult and theoretical. Neither group can see documentation as an easier, and in some sense more important topic than verification. To them, documentation is that "blah blah" that you have to write. In fact, unless we can solve the documentation problem, the verification work will be a waste of time.

In talking to people developing commercial software we find that documentation is neglected because it won't speed up the next release. Again, programmers and managers are so driven by the most imminent deadline, that they cannot plan for the software's old age. If we recognize that software aging is inevitable and expensive, that the first or next release of the program is not the end of its development, that the long-term costs are going to exceed the short term profit, we will start taking documentation more seriously.

When we start taking documentation more seriously, we will see that just as in other kinds of engineering documentation, software documentation must be based on mathematics. Each document will be a representation of one or more mathematical relations. The only practical way to record the information needed in a proper documentation will be to use formally defined notation.

29.8.2 Retroactive Documentation. A major step in slowing the aging of older software, and often rejuvenating it, is to upgrade the quality of the documentation. Often, documentation is neglected by the maintenance programmers because in their haste to correct problems reported by customers or to introduce new features demanded by the market. When they do document their work, it is often by means of a memo that is not integrated into the previously existing documentation, but simply added to it. If the software is really valuable, the resulting unstructured documentation can, and should, be replaced by carefully structured documentation that has been reviewed to be complete and correct. Often, when such a project is suggested, programmers (who are rarely enthusiastic about any form of documentation) scoff at the suggestion as impractical. Their interests are short-term interests, and their work satisfaction comes from running programs. Nonetheless, there are situations where it is in the owner's best interest to insist that the product be documented in a form that can serve as a reference for future maintenance programmers.

29.9.3 If It's Not Documented, It's Not Done. If a product is not documented as it is designed, using documentation as a design medium, we will save a little today, but pay far more in the future. It is far harder to recreate design documentation than to create it as we go along. Documentation that has been created after the design is done, and the product shipped, is usually not very accurate. Further, such

documentation was not available when (and if) the design was reviewed before coding. As a result, even if the documentation is as good as it would have been, it has cost more and been worth less.

Robert Glass. Facts and Fallacies of Software Engineering, Addison Wesley, 2003, pg 120.

In examining the tasks of software development versus software maintenance, most of the tasks are the same -- except for the additional maintenance task of "understanding the existing product." This task consumes roughly 30 percent of total maintenance time and is the dominant maintenance activity. Thus it is possible to claim that maintenance is a more difficult task than development.

Eick et al. "Does Code Decay?", IEEE Transactions on Software Engineering, January 2001.

- **Excessively complex (bloated)** code is more complicated than it needs to be to accomplish its task. If rewritten, bloated code could become easier to understand and simpler to maintain. The nesting complexity of a line of code is the number of loops and conditionals enclosing it.
- A **history of frequent changes** also known as code churn, suggests prior repairs and modifications. If change is inherently risky, then churn signifies decay.
- Similarly, code with a **history of code faults** may be decayed, not only because of having been changed frequently, but also because fault fixes may not represent the highest quality programming.
- **Widely dispersed changes** are a symptom of decay because changes to well-engineered, modularized code are local.
- **Kludges** in code occur when developers knowingly make changes that could have been done more elegantly or efficiently. That kludged code will be difficult to change is almost axiomatic.
- **Numerous interfaces** (i.e. entry points) are cited frequently by developers when they describe their intuitive definition of code decay. As the number of interfaces increases, increasing attention must be directed to possible side-effects of changes in other sections of code.

Barry Boehm. Software Cost Estimation with COCOMO II, Prentice Hall, 2000, pg. 23, 28.

The Maintenance Adjustment Factor (MAF), Equation 2.10, is used to adjust the effective maintenance size to account for software understanding and unfamiliarity effects, as with reuse. COCOMO II uses the Software Understanding (SU) and Programmer Unfamiliarity (UNFM) factors from its reuse model to model the effects of well or poorly structured/understandable software on maintenance effort.

Table 2.5 Rating Scale for Software Understanding Increment (SU)

	Structure	Application Clarity	Self-Descriptiveness
Very Low	Very low cohesion, high coupling, spaghetti code.	No match between program and application world-views.	Obscure code; documentation missing, obscure, or obsolete.
Low	Moderately low cohesion, high coupling.	Some correlation between program and application.	Some code commentary and headers; some useful documentation.
Nominal	Reasonably well-structured; some weak areas.	Moderate correlation between program and application.	Moderate level of code commentary, headers, documentation.
High	High cohesion, low coupling.	Good correlation between program and application.	Good code commentary and headers; useful documentation; some weak areas
Very High	Strong modularity, information hiding in data/control structures.	Clear match between program and application world-views.	Self-descriptive code; documentation up-to-date, well organized, with design rationale.

Ellen Ullman. "The Dumbing-down of Programming, Part II"

I used to pass by a large computer system with the feeling that it represented the summed-up knowledge of human beings. It reassured me to think of all those programs as a kind of library in which our understanding of the world was recorded in intricate and exquisite detail. I managed to hold onto this comforting belief even in the face of 20 years in the programming business, where I learned from the beginning what a hard time we programmers have in maintaining our own code, let alone understanding programs written and modified over years by untold numbers of other programmers. Programmers come and go; the core group that once understood the issues has written its code and moved on; new programmers have come, left their bit of understanding in the code and moved on in turn. Eventually, no one individual or group knows the full range of the problem behind the program, the solutions we chose, the ones we rejected and why.

Over time, the only representation of the original knowledge becomes the code itself, which by now is something we can run but not exactly understand. It has become a process, something we can operate but no longer rethink deeply. Even if you have the source code in front of you, there are limits to what a human reader can absorb from thousands of lines of text designed primarily to function, not to convey meaning.

Lauren Weinstein. "The Devil You Know". Communications of the ACM, December 2003, pg 144.

The underlying problem is obvious. Get past the flashy graphics and fancy user interfaces, and you frequently descend into a nightmarish realm of twisted spaghetti-like code that might better belong in a Salvador Dali painting. One recurring type of software security bug, buffer overflows, dates back to the dawn of computing. ... A plethora of patches will never be a substitute for true quality software.

Brian Foote and Joseph Yoder. "Big Ball of Mud", Fourth Conference on Pattern Languages of Programs, 1997.

A BIG BALL OF MUD is haphazardly structured, sprawling, sloppy, duct-tape and bailing wire, spaghetti code jungle. We've all seen them. These systems show unmistakable signs of unregulated growth, and repeated, expedient repair. Information is shared promiscuously among distant elements of the system, often to the point where nearly all the important information becomes global or duplicated. The overall structure of the system may never have been well defined. If it was, it may have eroded beyond recognition.

What does this muddy code look like to the programmers in the trenches who must confront it? Data structures may be haphazardly constructed, or even next to non-existent. Everything talks to everything else. Every shred of important state data may be global. Where state information is compartmentalized, it may be passed promiscuously about though Byzantine back channels that circumvent the system's original structure.

Variable and function names might be uninformative, or even misleading. Functions themselves may make extensive use of global variables, as well as long lists of poorly defined parameters. The function themselves are lengthy and convoluted, and perform several unrelated tasks. Code is duplicated. The flow of control is hard to understand, and difficult to follow. The programmer's intent is next to impossible to discern. The code is simply unreadable, and borders on indecipherable. The code exhibits the unmistakable signs of patch after patch at the hands of multiple maintainers, each of whom barely understood the consequences of what he or she was doing. Did we mention documentation? What documentation?

David Diamond. "For a Friend Assigned to a Maintenance Group" in Datamation, June 1976, pg 134.

The fellow who designed it
is working far away;
The spec's not been updated
For many a livelong day.
The guy who implemented it is
Promoted up the line;
And some of the enhancements
Didn't match to the design
They haven't kept the flowcharts,

The manual's a mess,
And most of what you need to know,
You'll simply have to guess.

David Parnas. "New Year's Resolutions for Software Quality", IEEE Software, January/February 2004, pg. 13.

We resolve to keep all program design documentation complete, precise, and up to date.