Egoless Writing: Improving Quality by Replacing Artistic Impulse with Engineering Discipline

Edmond H. Weiss, Ph.D.
Edmond Weiss Consulting
EdWeiss@aol.com
www.edmondweiss.com

Abstract

When technical communicators have a strong personal attachment to the publication they are preparing, this attachment may interfere with the design and testing of the publication itself. Documents developed by solo authors tend to be late, buggy, and exceedingly difficult for others to maintain. "Egoless" methods—collaborative and structured—break the proprietary connection between the writer and the book; in so doing they permit the most powerful tools of engineering and testing to be used. But they also reduce the satisfactions of the communicator's job.

I.7.1 Document Preparation—*document management*

Keywords*: documentation development, project management, collaboration, teamwork*

Part of the Problem...

There is a paradox in technical communication. Motivated, talented, intensely involved communicators can, strangely enough, be the greatest barrier to productivity and may even compromise the quality of the communication products.

In general, a strong proprietary attachment to the publication in progress results in an insulation of the work from outside scrutiny and testing. And, after a certain point, the emotionally engaged author becomes the protector and defender of the work—not only oblivious to most of its flaws but defensive toward anyone who suggests that they exist. Unable to separate the project from themselves they become "psychologically fused," in Baard's term, and instead of using the insights of others to improve the work, they "treat specific criticisms as general attacks" (Baard, 1994, p.15).

This problem was noted and discussed several years ago in connection with the productivity of computer programmers. "Egolessness" was a precursor of our current obsession with "team approaches." And the experiences of those who tried to redefine the programmer's profession bear directly on the future of technical communication.

ACM Journal of Computer Documentation, 2002,26:3-10

Writing and Programming

In many ways, technical communicators and programmers are alike. Moreover, several of the products of technical communication—such as manuals, training materials, and interactive databases—resemble computer programs in interesting ways. That is, technical publications (paper or electronic), which pass data and instructions to their readers, affect people rather in the way that software and programs affect computers and communication devices.

This analogy has led me to what I once called the Structured Documentation Hypothesis (Weiss, 1991, pp. 48-49): namely, that programs fail in many of the ways that publications fail and that, more important, the techniques that prevent programs from failing can be applied, with a little adaptation, to prevent publications from failing.

For example, programs and publications both benefit from modularity: chunking the project into small, manageable components. And both also benefit from "top-down testing": making sure that the high-level logic and interfaces work before developing the detailed code (text).

But these and other basic constructs of software engineering were not always the standard way of writing computer programs. (In many places they never have been.) In the early days of programming, before The Revolution [1], computer programs were typically the work of a single person, working alone for many weeks or months; the programs, though sometimes brilliant, were typically tangled, buggy, late, and generally different from what was promised. Usually, they were monolithic, not modular; untested, if not untestable; and generally incomprehensible to anyone other than their creator. (Rather like many technical publications from the 1970s and 1980s.)

This last problem—the opacity of the code and intractability of the logic to anyone but the solo programmer—led to the greatest difficulty of all. It was nearly impossible for other programmers to maintain or modify the work of the original programmer. And, since even the best of computer programs need fairly continuous fixing and adjusting, this lack of "maintainability" proved to be the most serious economic shortcoming in many programming organizations. Indeed, maintainability became the main variable in measuring the cost-effectiveness of software and applications. (Note that, like programmers, most technical communicators spend most of their time adapting and modifying existing information, rather than starting publications from scratch; thus, the maintainability or reusability of the materials can profoundly affect the cost of a writing project.)

In the early 1970s, then, a constellation of programming techniques emerged whose principal aims were to eliminate programming errors before the coding and, at the same time, generate a code that let itself be modified and fixed easily—even by someone other than the original programmer. Scores of important papers proved that better design, modularity, as well as the elimination of GOTOs and "pathological connections" (Yourdon and Constantine, 1979, p.40) would dramatically reduce the number of errors in the first version of a program and, later, facilitate any modifications [2].

"Egoless Programming"

The central problem in the implementation of these approaches, however, was NOT the techniques, but rather the programmers! Like most technical writers, early programmers tended to be solitary, "detached" to use Weinberg's (1971) term:

> There is no doubt that a majority of people in programming today lean in the "detached" direction, both by personal choice and because hiring practices for programmers are often directed toward finding such people. And, to a great extent, this is a good choice, because a great deal of programming work is "alone and creative" (Weinberg, p.15).
>
> Like most good things, however, the "detachment" of programmers is often overdeveloped. Although they are detached from people, they are attached to their programs. Indeed, their programs often become extensions of themselves...(Weinberg, p.53)

This deep personal attachment, the result of treating one's project as a “work of art” militates against formal engineering techniques, especially testing. Egoless programming is supposed to counter what King calls “the defensiveness of a solo programmer” (King, 1984, p. 11).

Weinberg continues:

> A programmer who truly sees his program as an extension of his own ego is not going to be trying to find all the errors in his program. On the contrary, he is going to be trying to prove that the program is correct—even if this means the oversight of errors which are monstrous to another eye. (Weinberg, p. 55)
>
> Programmers, if left to their own devices, will ignore the most glaring errors in their output—errors that anyone else can see in an instant. (Weinberg, p. 56)

Early on, then, Weinberg and others became enthusiasts for what these days is called the “team approach” to programming, even though, in most minds, programming was as ill-suited to collaboration as any task in the organization. “Egoless programming” was, in effect, just programming done by a team.

In its earliest incarnation, small groups of programmers would form egalitarian teams or work groups, without hierarchy. Although there was some chunking of the project into separate tasks, done independently, the emphasis was on collaboration—one person instantaneously commenting on or revising the work of another, everyone contributing code until the unique authorship of the program was lost and no one regarded it as his or her property.

These earliest attempts were somewhat chaotic, so that the egoless programming team eventually was forced to mature into a “chief programmer” team, with more structure and organization, but with the same collaborative spirit and the same goal: making the program the product of the group, rather than of the individual.

Egoless programming, even in the hierarchical team, is not necessarily fast or efficient. Frederick Brooks pointed out, only half satirically, that “adding manpower to a late software project makes it later” (Brooks, (1975), p. 25. But it does offer profound benefits, especially in the area of maintainability. Weinberg concludes:

> The adaptability of programs is also improved, for we are assured that at least two people are capable of understanding the program. Under certain programming circumstances, this represents an infinite improvement. Also, the entire work is less susceptible to being disturbed if one of the involved programmers happens to be... missing. This not only reduces variations in schedule, but also makes it more likely that at some time in the future, when the code must be modified, someone will be around who knows something about it. (Weinberg, p. 59)

According to Pressman (1982), this approach, “eliminating ego attachment to the software,” fosters more “thorough review” and “increased learning through side-by-side work” (Pressman, p. 89).

Involving several people in the project, then, increases the chances that at least one of the creators will always be around to solve problems and make changes. But, even if no one from the original team is available, there will still be the outlines, specs, notes, and minutes... the logical underpinning of the project, which is so helpful in modifying an existing program (or publication). These are the materials that most “artist” creators, working alone, do not create at all or will not share.

Document Quality versus the Writer’s Ego

Even if many programmers have learned to be team players, most technical communicators still have not. It is safe to guess that the least regimented and most egoistical employees in companies are the writers. And even workers who are usually disciplined and team-oriented tend to become artist-like when the assignment is to produce a long publication.

Everyone who has worked in technical communications for any length of time will share Weinberg’s opinion about the incompatibility of “ego attachment” and the normal requirements for editing and testing publications. Years ago, I half seriously dubbed this phenomenon The Principle of Enamorment: ***At a certain point in any large-scale writing project, the author falls in love with the work in progress***. After this moment of enamorment has passed, the artist-author, “fused” to the document, is incapable of receiving criticism well and is more likely to resist needed changes than to act on them. If usability testing comes after this moment, the test is likely to be disingenuous and its results are unlikely to be used.

Manuals and publications written by one person, working alone, without the obligation to produce testable intermediate outlines or models tend to be

- filled with technical errors
- delivered too late for thorough editing or testing
- inherently unmaintainable by anyone other than the solo author.

In writing, as in all large-scale creative projects, and even more than in programming, there is an inevitable clash between the ego and temperament of the artist/author and the needs of the project. This struggle is between the **artistic impulse** (privacy, lack of regimentation, lack of intermediate products and reviews, refusal to be judged by less knowledgeable persons) and the **engineering discipline** (problem-solving through public models and successive approximations). An artistically executed publication, while it may satisfy the personal needs of the author, and while it may even win awards from organizations of technical communicators, is also likely to be low in quality and very low in maintainability.

Clearly, there are two broadly different ways to write a manual or other document. The first, the artistic, is to compose it, crafting the sentences and paragraphs while they are being written, as would a "writer" working on a script. The second is to engineer it, preparing a series of increasingly finer specifications until, at last, a manual "drops out."

When the general public thinks of "writers," of course, they tend to think of the "artist" slaving over the sentences in a manuscript; there are bursts of inspired scribbling, followed by intense editing and reworking, interrupted by long waits for the next inspiration. In this stereotype, the writer works with nothing but paper and a writing instrument—few notes, no plans, no models or mockups. There are just the blank pages (or screens) and the writer's mind.

(There is a similar stereotype for computer programmers: people who think with their hands on the keyboard—trial and error, inspired guesses, flashes of genius...In movies and TV shows, programmers never consult a dataflow diagram!)

Interestingly, programming and manual-writing are two of the very few complicated, error-prone projects that can sometimes be carried off in this loose, undisciplined "artistic" style. No one would manufacture a car that way, or build a bridge by trial and error. Certainly, no one would write the script for a $100 million dollar movie that way either. Indeed, computer programmers and technical communicators are among the very few professionals who would, without hesitation, invest six person-months of effort on a nearly unspecified project and hope for it to turn out well.

Moreover, most writers of manuals, including more than a few professionals, work with even less specification than computer programmers, who generally work with less specification than the people in any other technical profession. To write a long book from an ordinary outline—an outline that uses the same conventions the writer learned in grade school—shows little appreciation for the virtues of engineering and design.

Preparing an Egoless Manual

The engineered manual or publication is the end-product of a deductive process that begins with an analysis of the knowledge climate of the product (that is, who needs to know what) and moves through a series of successive approximations to the individual pages/modules. Unlike other methods that use the name, it is truly **top-down**: it tests the interfaces first, and it integrates from the top, not the bottom.

The process of developing an egoless manual strongly resembles the process of developing a system or application:

- A high-level team clarifies requirements and develops a proposed set of publications.
- For each publication, a mid-level team develops detailed outlines, specifications and storyboards.
- Specifications and outlines are tested in a "walk-through."
- Working from approved designs or storyboards, individual writers provide small, well-defined pieces of text and illustration.
- The material is compiled, reviewed and edited by the design team.
- An independent person or organization tests the publication for accuracy and usability; the egoless team is not upset by the test findings and, therefore, acts on them.
- The modules developed for the publication are stored in a document database and reused in later publications.

This method was (and is) a great shock to the egotistical artist-writer, who puts relatively little effort into planning. Instead, the egoless, engineered model puts most of the effort into the planning—definition, design, and modeling—with the next greatest effort in testing. Drafting—where artists put their main talents—is merely the implementation of the design, not the creation of the product.

	Artistic Model	**Engineering Model**
Accessibility of Work	•Private •Intuitive, Personal	•Public Models •Public Access
Process	•Impulsive Solutions •Improvisation	•Formal Process •Successive Approximation •Spec-Model-Test
Progress	•All-or-Nothing •Latest Delivery	•Intermediate Products •Intermediate Testing
Standards	•Intuitive, Personal •Self-Indulgent	•Normative Standards •Acceptance Criteria
View of the Reader	•Independent •Resourceful	•Dependent on Designers •Error-Prone

Figure 1: The Artistic Model versus the Engineered Model

As Figure 1 shows, these two approaches view the process and product of technical communications in almost entirely different ways. The engineered publication is accessible and discussible, even at early stages of planning. In contrast, artist-writers do not want to show the work until it is “ready.” They tend to be unable to describe it until it exists, and, once it exists, will defend it against all critics and test results. Usually, no one but the artist gets to review, test, or criticize the work of art until it is almost finished, much too late to make major changes. In contrast, the engineered manual is discussed extensively—and criticized and revised—at several intermediate stages, before the author's ego is too deeply invested in the work.

Another key contrast is the *process*. If preparing manuals is an art, then the creativity is in the drafting, the impulsive composing of the words and sentences. In contrast, if a manual is an engineered product, then the creativity is in the *engineering*, writing the specifications, building and testing models—all of which precedes the execution of the design (the draft).

From a manager’s perspective, the main difference may be the way *progress* is observed and measured. Publications developed in the egoistical, artistic way tend to be late, and without substantive intermediate products. In effect, they cannot be edited, tested, and evaluated until they are completed—too late for significant changes, often too late for even rudimentary corrections.

The two models also suggest contrasting criteria and *standards*. If a publication is art, then the basic criteria are style and appeal—an intuitive sense of correctness and craft, peculiarly understandable to the writer but difficult to explain to others. If it is an engineered product, though, the basic criterion is whether it meets the specifications, performs the job it was assigned for. And the advanced criteria also change. For artists, a very good book is one that has beauty, elegance, “class.” If a book is an engineered product, though, the *advanced criteria* are from engineering: maintainability (how easy it is to update and enhance the book) and reliability (how often the book “fails” in use).

The *view of the reader* is also in dispute. The artist views readers as independent and active; the burden is on the readers to find things and apply them correctly. If books are engineered, though, readers are less independent. Instead, they rely on the design of the book; the burden shifts to the technical communicator. In this view the writer controls the attention of the readers, protecting them from error, much as software controls hardware.

Egoless Writing without Manuals

The egoless and collaborative writing models of the 1980s met resistance from those whose greatest pleasure was the solitary task of creating an interesting, innovative manual *ex nihilo*. For many writers, working in teams and developing publications through stepwise approximations destroyed the core motivation of the job, the central satisfaction of the profession. (In some companies, my seminars on “structured documentation” met with outright hostility from experienced technical writers.)

But nothing in the 1980s could have prepared the technical communicator for the ego-obliterating model of the 1990s. Recall that, in the egoless, engineer-like approach just described, the team organized to produce a publication, a book. At the very worst, the individual writer was forced to share in the accomplishment of the finished product. Although there may not have been the old artistic satisfaction, there was at least the pride of participation in a team or corporate project.

How much less pride there is in contributing to a document database: a repository of small articles of text and picture, which will be assembled or invoked into a variety of online and paper publications. In the 1980s, technical communicators produced modular publications, whose modules were then stored in a database and reused in other publications. In the preferred 1990s method, however, the technical communicator does not so much create publications as directly feed the database, in which all potential publications reside "virtually."

It is scarcely an exaggeration to say that, at the cutting edge, technical communicators are no longer creating well-defined books and publications at all. (This observation is from the standpoint of the history of ideas and technology. Just as there are still programmers writing unstructured COBOL applications, there are still many technical communicators writing unstructured artistic publications, as well as 1980s-styled modular manuals.) Today's technical communicators, even if they still believe they are writing manuals, are really contributing text to a multimedia database, from which intelligent users, or naive users aided by intelligent software, will extract selected items for their support. (Or, alternately, professional communicators will *generate* ad hoc publications from this database.)

Ironically, this dramatic empowerment of the readers and users of documentation has led to a potential enfeeblement of the technical communicators, not merely a loss of vainglorious ego but even of basic self-esteem. For the first time in the history of technical communication, the senders of the message are being asked to yield control over such decisions as the sequence and timing of information, as well as such basic questions as the design and layout of pages or screens. In what sense has the "writer" of a hypertext book or a Windows help file actually created a real communication product? Which of the 1000s of "virtual" publications was actually written by the technical communicator? Or by anyone else, for that matter?

To serve the objective of universal, hardware/software-independent documentation—materials that can be printed, displayed, or even speech-synthesized on any system—the new model even militates against letting the technical communicator control the layout and appearance of pages. Such technologies as SGML, HTML, and XML underscore the sudden abrupt change in the power of the technical communicator: from perfect control over every element of the page to provider of "tagged input." (The control of the output resides elsewhere in Data Type Definitions in other, unseen computers.) Not only does this change produce scores of practical communication problems, such as not being able to guarantee that a Warning is on the same page or screen as the dangerous procedure, it also destroys any last vestige of ego involvement by the author. Increasingly, the relationship between the publication and the technical communicator will resemble the relationship between the gold wedding ring and the gold miner.

Interestingly, the underlying logic of the new "virtual" documentation is that it is not just the ego of the solitary writer that interferes with the needs of the readers/users/receivers: it is the inertial force of the fixed, "real" book itself. That is, individual books, with their tyrannical fixed sequence and author-imposed design, are incompatible with the aims of the new technical rhetoric, in which messages are not so much transmitted by senders as extracted by receivers. (In the 1980s, technical communicators tried to organize manuals so that readers would do as little skipping and looping as possible; today, with hypermedia, the issue is moot.)

Also inherent in this approach is the ego-threatening notion that the technical problems of user documentation have been largely solved. Experienced developers and technical communicators know by now the several workable ways to explain and teach procedures (any of them will do), so that a new piece of software ***suggests*** the kinds of documentation modules it needs. That is, after nearly fifteen years of supporting a variety of system users, the industry now understands the problem of user support and can offer adequate (if not brilliant) documentation and training using standardized forms and styles. (Indeed, for those still producing actual manuals, Microsoft Press and Maran Communications—among others—provide scores of publications that could be implemented as "best practices" or even templates by all software writers.)

In effect, those hundreds or thousands of technical communicators currently doing research and development on the enhancement of "usability" are merely making slight adjustments at the inefficient margins of communication effectiveness, probably not beneficial enough to users to justify the cost of the research. One even sees an occasional suggestion in the military's CALS program that technical writers do not really add value to the material, that users would be just as well off with direct access to the engineering files of the systems [3].

Egolessness and Professionalism

Wherever egoless methods have been introduced, in whatever discipline, usually they have been supported by leaders and managers—those concerned with overall efficiencies and hard measures of performance—and resisted by those whose egos were judged part of the problem. This should have surprised no one. The claim that a complicated project can be performed by fortuitously assembled teams of interchangeable workers is equivalent to the claim that the project is intellectually trivial. All artists know that there are ways to "manufacture" cheap products that superficially resemble works of art; yet, no artist can be expected to accept graciously the notion that the manufactured version is high enough in quality to rival the artist's version.

Whenever people with an ego-stake in their craft are told to limit their independent choices, they sense, correctly, that their talents and uniqueness are being devalued. Perhaps the programmer believes that GOTO is the best choice. Perhaps the technical writer believes that a certain If-Then instruction should be written Then-If. ***An artist cannot be wrong***. But a clerk, operator, or technician can be—often is—wrong.

Although many of the egoless techniques of the software engineering revolution have been adopted in sophisticated programming organizations, it would be an overstatement to say that they had become commonplace. Indeed, the resistance of the American programmers to nearly every new tool and productivity advance in the past 20 years has placed them at great risk, according to Edward Yourdon (1992), who expects that an increasingly larger share of America's programming will eventually be "outsourced" to places like India, where the productivity techniques have been embraced. Yourdon predicts that our international competitors will work harder, for longer hours, with considerably less pay, and with *a more sustained commitment to software quality than their American counterparts*. (Yourdon, pp. 1-18) And, most interestingly, Yourdon found that the typical American programmer (like the American autoworker of the 1960s) simply cannot believe that this is true!

Similarly, the current epoch of egoless technical communication could be staffed with people far less talented than the current cohort of North American technical communicators. The intellectual leadership in this field developed their skills and reputations when the manual was still an interesting technical issue, when people still regarded the writing of instructions for electronic devices as a set of challenging research problems. And it is hard to escape noticing that many of the students currently enrolled in graduate programs are unaware that most of the problems have been solved and the profession has been dramatically altered.

As in programming, technical communications will always need a small cadre of inventive, imaginative writers and designers, to organize and manage the media databases. But most of the writing and illustrating, the input to these databases, will (should) be provided by a less professional, less talented, less expensive class of workers. (Even recurring problems in English style will eventually be corrected by the use of fixed vocabularies and limited sentence forms [so-called Controlled English], as well as by style-correcting software, allowing more of the work to be done by people for whom English is a second language.)

So, technical communicators have some interesting choices to make in this decade. They can continue to resist egoless methods, swim against the tide in pursuit of personal satisfaction. Or, they can aspire to become the central cadre of inventors and designers, learning to delegate the "writing" to the miners, here or off shore. Or they can become "knowledge workers," people whose job is less the generation of information than helping people to find it, use it, adapt it, or deal with the frustrations of a workplace rich in data but poor in insight.

References

Baard, P. (1994). Psychological fusion and personal conflict in organizations. *IEEE Transactions on Professional Communication*, 37 (1).

Brooks, F. (1975). *The Mythical Man-Month*. Reading, MA: Addison-Wesley.

King, D. (1984). *Current Practices in Software Development*. New York: Yourdon Press.

Pressman, R. (1982). *Software Engineering: A Practitioner's Approach*. New York: McGraw-Hill.

Weinberg, G. (1971). *The Psychology of Computer Programming*. New York: Van Nostrand.

Weiss, E. (1991). *How to Write Usable User Documen-*

tation (2nd ed.). Phoenix: Oryx Press.

Yourdon, E. (1992). *Decline & Fall of the American Programmer.* Englewood Cliffs, NJ: Prentice-Hall.

Yourdon, E. and Constantine, L. (1979). *Structured Design*. Englewood Cliffs, NJ: Prentice-Hall.

Notes

(1) The upgrading of the programmer's job from craft to engineering has been called a "revolution"; see, for example, Edward Yourdon, ed., Writings of the Revolution, New York; Yourdon Press, 1982.

(2) For a collection of the seminal papers on this subject, see Edward Yourdon, ed., Classics in Software Engineering, New York; Yourdon Press, 1979.

(3) A JCALS paper speaks of "users [who] will be relieved of the burden imposed by the paper-intensive environment."