CHAPTER 1

Collaborative Writing, Software Development, and the Universe of Collaborative Activity

DAVID K. FARKAS

In recent years, collaborative writing has become a significant area within the discipline of composition studies, and it is now receiving a good deal of attention. We should remember, however, that collaborative writing is not simply a form of writing. It is also one of innumerable forms of human collaborative activity. Just as people collaborate to prepare documents, so do they collaborate to build bridges, climb mountains, govern communities, and perform innumerable other tasks.

Much of the time collaborative activity is carried out transparently and with little or no difficulty. A husband and wife prepare the family breakfast and have no thought that they are engaged in a collaborative task. Other collaborations are complex and difficult affairs in which the need to coordinate the activity of a group of individuals is a prominent consideration. One of the broad theses of this chapter is that collaborative writing is one of these complex and difficult forms of collaboration.

The second broad thesis of this chapter is that collaborative writing has much in common with other forms of collaboration. All forms of collaboration, for example, require participants to develop a common understanding of their goal

and methods and to coordinate their efforts toward achieving that goal. But, of course, collaborative writing is much closer to some classes of collaborative activity than to others.

The third broad thesis is that we can learn a great deal about collaborative writing by looking at other forms of collaborative activity, both by observing it directly as it occurs all around us and by adopting and adapting insights and methodologies from disciplines that in various ways address the problem of human collaboration.

My procedure in this chapter is

1. to demonstrate, by means of comparisons with other collaborative activities, why collaborative writing is difficult and to point out that the difficulties in collaborative writing are not unique but rather are shared with broad classes of collaborative activities, and
2. to look closely at one particular collaborative activity–software development–and show how software development can contribute to the understanding and improved practice of collaborative writing and what it has contributed already.

Before proceeding further, I will offer a definition of collaborative writing that consists of four basic forms of writing, along with all possible combinations:

1. two or more people jointly composing the complete text of a document;
2. two or more people contributing components to a document;
3. one or more persons modifying, by editing and/or reviewing, the document of one or more persons; and
4. one person working interactively with one or more persons and drafting a document based on the ideas of the person or persons.

While this chapter applies in at least a general way to all forms of collaborative writing, I most often refer to and I implicitly assume a combination of Forms 2 and 3, a situation in which two or more persons contribute components of a document and one or more persons review and/or edit the document. I de-emphasize Form 4 because it is a somewhat marginal instance of collaboration in that one person is doing all of the actual drafting. I de-emphasize Form 1 because it is less often practiced than the other forms of collaborative writing. Joint composing vastly reduces writers' productivity; the two (or more) writers perform the work of one and, in fact, often produce less material than one writer would. Also, joint composing frequently results in a great deal of disagreement and irritation over individual choices in diction and syntax. This form of collaborative writing is, in fact, primarily useful in situations when the document is short, when agreement on exact wording is crucial, and when there is need to eliminate what would otherwise be an extended round of reviews and edits among the collaborators.

WHY IS COLLABORATIVE WRITING DIFFICULT?

Collaborative writing *is* difficult. I think our collective experience tells us that. Also, the literature supports this idea [1, 2]. Below I offer six reasons why:

1. Highly integrated documents are very complex artifacts.
2. The process of preparing a document becomes more complex when it is performed collaboratively.
3. The writing process generates strong emotional commitments.
4. Documents are reworkable and are subject to infinite revision.
5. Collaborative writers lack fully adequate terms and concepts with which to create a clear and precise common image of the document they wish to produce.
6. It is difficult to predict or measure success.

These six reasons are derived only on an ad hoc basis, not from a comprehensive theory of collaborative writing. None exists. Others studying collaborative writing, therefore, can create a somewhat different list. This list, however, as it is developed below, does articulate much of the difficulty in collaborative writing. In addition, it reveals numerous points of contact between collaborative writing and other collaborative activities.

1. Highly Integrated Documents Are Very Complex Artifacts

Obviously, not all the artifacts produced by collaborative work are complex. A group of children, for instance, will make a crude snow figure consisting of little more than three balls of snow. Any document, however–even a short, routine business letter–is a moderately complex artifact, a web of words that have been chosen on the basis of syntactic, semantic, and pragmatic considerations.

Greater complexity begins to come in longer documents that convey large amounts of information. A requirement for truly great complexity is the high degree of integration–the myriad and complex linkages–among the components of a large document. This integration can take the form of a carefully developed theme or a carefully orchestrated emotional impact that builds and modulates through a long document. It can also take the form of an elaborate system for presenting reference information–for example, a sophisticated reference manual that at each topic in the document directs the reader to all the other places where related information is to be found.

In contrast, there are documents characterized by fairly discrete components and, hence, a "loose" form of collaboration. This category includes a technical report containing sections individually researched and authored by scientists who only coordinated on a few matters. While each section bears upon the overall topic in an appropriate way, the format, organization, depth of treatment, and perhaps even the approach differ from section to section. Such documents

Ensemble music is a collaborative activity in which the medium, like language, is subtle and elusive. Also, apart from the technical aspects of music, the terms and concepts musicians work with are highly metaphorical and lacking in precision. Most important, each musician, while engaging in a very personal and expressive form of communication, must blend his or her individual voice into a group product. In the case of orchestral music, the score largely constrains the choices of the individual musicians, but a conductor is still employed to coordinate aspects of the musicianship. In the case of improvisational music, such as jazz, each performer proceeds with considerable autonomy but must maintain ongoing communication with the others. The very frequent success enjoyed by ensemble musicians of all kinds indicates that there is much in their methods and traditions to be carefully examined by those with an interest in improving collaborative writing.

Another collaborative activity that should be studied is software development along with its associated discipline, software engineering. In this instance, however, I will do more than glance at its relevance to collaborative writing. Rather, the nature of the contribution that software development can make, and has already made, is the subject of the second part of this chapter.

SOFTWARE DEVELOPMENT

Software Development and Collaborative Writing

Although computer programs are produced by individual programmers, most commercial software is sufficiently large to require a team of programmers. Software development, therefore, is by and large a collaborative activity.

As a collaborative activity software development has significant similarities to collaborative writing. First of all, programmers, like writers, create long and complex strings of language—computer code. The language of programmers, of course, is a formal rather than a natural one, and its primary "audience" is a group of electronic components. But the electronic components, like human beings, make demands upon the language they deal with, and because the components require logic and coherence, each programmer's part must fit properly into the whole. Furthermore, computer code must be readable not only by electronic components but by human beings as well. Otherwise, it is impossible for members of programming teams to work with each other's code or for finished programs to be maintained by other programmers at a later date. Programmers, in fact, speak about good and bad programming "style" in somewhat the same way that writers do [13, 14].

Software development also faces many of the same difficulties that collaborative writing does. In fact, if we adapt the list of six difficulties in collaborative writing to software development, we get a meaningful picture of the similarities and differences between the two.

1. Large computer programs are very complex artifacts and are highly prone, at least initially, to failure.
2. Because large computer programs are produced collaboratively, the process itself is extremely complex and fraught with difficulty. The work, first of all, must be decomposed among the programmers, and action must be taken continually to correct for divergences from the common image. In contrast to writing, software project leaders can specify this image precisely, but, as Edward Yourdon notes, wherever the specification is incomplete, individual programmers will tend to make individual design decisions [15, p. 150]. Furthermore, as Yourdon notes again, "communication problems between programmers become unmanageable on large projects" [15, p. 150].
3. Programmers, like writers, form strong emotional commitments to their work. They become committed to their own programming techniques and the code they produce. Furthermore, they tend to regard their segments of code as "private masterpieces" rather than as part of a group product [15, p. 172].
4. Like writing, code is infinitely reworkable and like writing is subject to successive reworking. Managers and customers change specifications, and developers must struggle to keep the project on schedule and within budget and to maintain conceptual integrity [16, pp. 151-240].
5. In clear contrast to collaborative writing, software development enjoys precise terms and concepts. This precision extends in many cases to the language of mathematics. These terms and concepts enable software developers to clearly and precisely specify what they wish to create.
6. In contrast to the situation faced by writers, software developers receive feedback that is objective. This feedback, however, is only somewhat more readily obtainable than the feedback writers seek through usability testing. If there are major flaws in the coding, if–let us say, serious incompatibilities have been introduced at the last minute–these flaws become readily apparent, because the program will crash. On the other hand, subtler problems conceal themselves deep within the code; and if not discovered and corrected through a process of rigorous software testing, they will reach the user in the form of a "buggy" product.

The Discipline of Software Engineering

The difficulty in creating large computer programs became increasingly apparent in the 1960s, and led to the emergence of a new discipline devoted to the support of software development. This discipline is software engineering. Utilizing engineering methodology, management theory, and findings from psychology, software engineers attempt to improve the efficiency with which computer programs are designed, written, tested, and maintained. Because inefficiencies in large software development projects are so expensive and so visible, software engineering is an active, well-funded discipline.

Because software development is a collaborative activity, software engineering is in large part the study of human collaboration in a particular setting. So, for example, Frederick Brooks' *Mythical Man-Month* is both a pioneering work in the field of software engineering and a classic study of human collaboration [4]. As we will see, collaborative writing has already benefited significantly from some of the work done in software engineering, and, can benefit still more. The aspects of software development and software engineering most relevant are 1) models for software development, 2) tools for collaboration, 3) metrics, and 4) visibility and access to resources.

Models for Software Development

Programming teams employ conceptual models to develop software. These models have close analogs to collaborative writing. The classic model is "top-down design." The first steps are to ascertain system requirements, perform top-level design, and then perform successively more detailed design. When this planning process is complete, programmers write their components of the program. The completed components are then integrated into a whole, and the program is tested in order to determine if further work is necessary. One criticism of this model is that it remains a set of abstract plans for too long. Consequently, many developers modify the model by calling for "draft" versions of each component midway through the development process. These components are tested and evaluated, so that more "proven" versions of each component are available for integration into the whole [17, pp. 22-25].

"Rapid prototyping" is a model that requires less detailed preliminary planning. Instead it calls for the rapid and inexpensive development of a partly functional prototype that has the "look and feel" of the finished product. This prototype is evaluated by users and then refined by the programmers in successive cycles until the program is complete. The advantage of this more casual approach to software development is that users get to critique early versions of the program before the design has solidified [18, pp. 26-27].

Another model is "structured design" [15, pp. 86-100]. Its underlying concept is "modularity." Modularity, however, is a broad concept that pervades many models of software development. The idea behind modularity is to reduce the overall complexity of a program by breaking it down into a set of semi-discrete units whose relationships with one another are fairly easy to define. This contrasts to "spaghetti code," in which any segment of code can have connections to any other segment.

Other models focus upon the collaborative arrangements of the development team. Among these is the "chief programmer team" model. Here one very superior programmer does all the critical programming on a project, but has a very complete support staff—assistant programmers, an administrator, software testers, and so forth—that enables her to get a large job done in a reasonable

length of time. The idea is to reduce the complexity of the software development process by drastically reducing the amount of communication and coordination that is necessary among team members and to ensure that at least one person has a complete grasp of the project. One major problem with the model, however, is that only relatively small projects can be undertaken with this method [4, pp. 29–37].

A very different model focusing on collaborative arrangements is that of the ego-less programming team. Here the goal is to achieve synergism and to eliminate the problems caused by emotional commitments, including the tendency to view one's work as a "private masterpiece." In an ego-less programming team, for example, team members have free access to each other's code. In addition, objective discussion of each person's work is formalized through the use of special meetings known as "walkthroughs." The group rather than individuals assumes responsibility for the success of the project, and the emphasis is on the achievement of the team rather than individual contributions [15, pp. 171–172].

All of these models have analogs in writing. Top-down design is akin to having a writing team develop and agree upon the top-level entries of an outline and then proceed to lower-level entries. It might include drafting preliminary versions of chapter introductions and other high-level components. One conception of how to apply rapid prototyping to the domain of writing is for the writers to produce a quick rough draft or even a dictaphone recording of representative sections of the document and solicit comments from managers or trial readers before refining the design.

Modular documents do exist—catalogs and collections of abstracts are examples. Indeed, although modularization can only be used in certain communication situations, the approach simplifies the collaborative process vastly, so much so that early on in this chapter modular documents were classified as a nonproblematic form of collaborative writing.

There have no doubt been innumerable instances in which collaborative writing has been carried out very approximately along both the model of the chief programmer team and the ego-less programming team—certainly there are times when it is desirable and feasible to simplify collaborative writing using the marginal Form 4 described at the beginning of this chapter, and certainly it is often desirable to attempt to inculcate team spirit among the group of writers and to appropriately channel the strong emotional commitment generated by the writing process. But here is the key point: because of the work of software developers and software engineers, these software development models have been carefully formulated and debated, systematically applied, and empirically studied. There is a highly analytical literature that describes these models and their underlying concepts and explains and debates their merits and deficiencies. On the other hand, the analogs in collaborative writing are far less carefully developed and less often studied. They exist primarily in the collective experience of individual writers and writing team leaders. The literature is scant and very often anecdotal.

Collaborative writing may require its own distinct development models, but there is certainly much to be borrowed from the models of software development and the analytical literature that accompanies them. Of note in this regard are Ronald Guillemette's rationale for adapting rapid prototyping to documentation writing [19] and Edmund Weiss' plan, which incorporates a variety of software engineering concepts, for preparing computer documentation using a modular format that both satisfies the information needs of readers and simplifies the collaborative writing process [20].

Computer Tools for Collaboration

Because collaborative writing and software development are very complex processes, both writing teams and programming teams face the challenge of keeping track of what has been produced, ensuring that changes are made only by those with authorization, and making the most current version of the product (as well as earlier versions) available to all who need to view it. Software developers are tool-builders, and since the 1960s they have been creating and refining a whole class of tools for this purpose [4, pp. 132-133]. These "configuration management" tools are the computer equivalent of a project librarian. If, for example, a programmer wants to see the most recent version of a segment of the project code, the computer delivers it, indicates by whom and when the last revisions were made, and can even display just those lines in which changes were made.

These tools have now migrated into the world of technical publications, and the most sophisticated electronic publishing systems now possess elaborate configuration management capabilities [21, 22, 23]. These capabilities are beginning to appear on microcomputers, both as advanced features of high-end word processors or as separate products. In this form, they will soon be available to almost everyone who engages in collaborative writing.

Software developers, especially those working in research environments, have also devised a broader range of tools to support their activities and to advance our understanding of collaborative work. There are very interesting noncommercial systems which facilitate face-to-face meetings [24, 25] and which facilitate meetings among people working at different locations [26, 27]. There are also systems designed to facilitate collaborative writing in new ways. Neptune, for example, is an experimental hypermedia system that enables the members of writing teams to share text across a network and, more significantly, to manipulate and view this pooled material in ways that help them visualize the emerging document [28]. The practice of collaborative writing will certainly improve as these tools are refined and commercialized. In the meantime, the computer science literature contains articles describing these tools, the theoretical assumptions they embody, and the findings of experiments that involve their use. This literature provides valuable insights concerning both collaborative writing and collaborative work in general.

Development of Metrics

Software engineers have developed metrics, appropriate measurements, that they use to better understand and plan their work. A very crude but useful metric is the size of the project measured in lines of code. It is helpful, for example, to know that the program you are about to produce is estimated at 20,000 lines of COBOL code. Another crude metric is programmer productivity measured in lines of code per unit time. The inadequacy of these metrics, however, is easy to see: one 20,000 line program may be much more complex to write than another, and one programmer may write 300 lines of easy code in a day whereas another may produce 50 lines of very difficult code. Software engineers, therefore, have developed very sophisticated, often highly mathematical, metrics that give them a much more precise understanding of the nature of a software development project [29, 30].

Writing teams understand and use both the metric of document length and the productivity metric of pages produced per unit time. But we need to develop more sophisticated metrics in order to better plan and schedule writing projects and to distribute the workload more evenly and with more attention to the special abilities of individual writers.

We would benefit from predictive metrics that measure the writer's familiarity with the source material or the amount of time the writer will spend getting information from technical experts. Other predictive metrics might measure structural characteristics of an unwritten document, based on an outline or an understanding of its organization. For example, a metric that indicated the degree of interrelatedness of the parts of a document would help predict how quickly the writing would go, what the likelihood was of leaving out necessary information, how the document should be decomposed among a team of writers, and which components should go to the most capable members of the team.

Software engineering has something to contribute to the development of sophisticated metrics for writing. To begin with, we can look to software engineering for hints as to what sorts of metrics are desirable and how to develop them. The applicability of software metrics is, of course, limited by the difference between formal and natural language and the fact that the primary audience for code is not a human being. Consequently, metrics for writing will be based largely on concepts of language and discourse that will be drawn from linguistics and rhetoric and an understanding of composing processes and reading processes that will come from psychology. But software engineering metrics may provide the mathematical relationships that turn distinctions about language and an understanding of psychological processes into useful quantitative measurements.

Visibility and Access to Resources

Although professionals in many fields do some incidental programming, almost all large software development projects are created by full-time programmers. Software development, therefore, has become a distinct profession.

Furthermore, the profession has been successful in achieving visibility and gaining access to resources. Corporations and government agencies fund research in software development, and indeed the discipline of software engineering has emerged to support this activity. Corporations and other producers of software have recognized the importance of programmer productivity and quality software, and hire consultants and seminar instructors to help them create an appropriate environment for software development. Books such as Paul Licker's *The Art of Managing Software Development People* serve the same function, explaining to managers that programmers have specific needs and require specialized management structures and procedures [31].

On the other hand, although collaborative writing is an all pervasive activity practiced throughout the professional world, it is largely invisible. This is because it is primarily practiced as an ancillary activity by bankers, marketers, chemists, social workers, and so forth. In many instances, the wasted time and energy, the animosities that have arisen, and even the poor quality of the finished document are forgotten shortly after the project has ended. The full-time collaborative writing professionals, primarily technical communicators and other corporate publications professionals, make up only a small contingent. Furthermore, in the nation's English departments, the primary home for the study of writing, the concern with collaborative writing is only recent.

The main theme of this chapter is that collaborative writing is difficult, that its difficulties are largely shared with other activities, and that we can look at these activities and profit. Looking at software engineering, one thing we learn is the need to teach about collaborative writing as well as study it. We need to ensure that writing teams have adequate visibility and access to resources.

Corporations should come to realize that writing, especially in groups, is a complex and delicate process and that special management procedures are necessary. For example, the document review process should be conducted with sensitivity to the burden it places on writers, and special procedures, such as some variation on the ego-less programming team model should be adopted. Corporations should also recognize the need for special resources, such as specialized computer equipment and the time and money necessary for usability testing. Finally corporations and government agencies should realize the value of well-funded research in the area of collaborative writing.

Academics and practitioners working together should be able to achieve a noticeable effect in the corporate world. But whatever our degree of success over the near term, the recent trend toward emphasizing collaborative writing in the schools is very promising. If this trend continues and strengthens, we can expect future generations of professionals to understand more about the practice of collaborative writing and to give it greater support within their organizations. This is important, for it will help create a more productive and congenial workplace.

REFERENCES

1. L. S. Ede and A. A. Lunsford, Why Write . . . Together: A Research Update, *Rhetoric Review, 5*:1, pp. 71–77, 1986.
2. J. Paradis, D. Dobrin, R. Miller, Writing at Exxon ITD: Notes on the Writing Environment of an R&D Organization, in *Writing in Nonacademic Settings,* L. Odell and D. Goswami (eds.), Guilford Press, New York, 1985.
3. J. Gall, *Systemantics: How Systems Work and Especially How They Fail,* Pocket Books, New York, 1976.
4. F. B. Brooks, Jr., *The Mythical Man-Month: Essays on Software Engineering,* Addison-Wesley, Reading, Massachusetts, 1975.
5. D. K. Farkas and N. J. Farkas, Manuscript Surprises: A Problem in Copy Editing, *Technical Communication, 28*:2, pp. 16–18, 1981.
6. D. K. Farkas, The Concept of Consistency in Writing and Editing, *Journal of Technical Writing and Communication, 15*:4, pp. 353–364, 1985.
7. E. Gold, Don't Let the Approval Process Spoil the Book, *Simply Stated, 49,* pp. 1–2, September 1984.
8. G. M. Schumacher and R. Waller, Testing Design Alternatives: A Comparison of Procedures, in *Designing Usable Texts,* T. M. Duffy and R. Waller (eds.), Academic Press, New York, 1985.
9. P. Wright, Is Evaluation a Myth? Assessing Text Assessment Procedures, in *The Technology of Text,* Volume 2, D. H. Jonassen (ed.), Educational Technology Publications, Englewood Cliffs, New Jersey, 1985.
10. J. M. Lauer and J. W. Asher, *Composition Research: Empirical Designs,* Oxford University Press, New York, 1988.
11. J. Syer and C. Connolly, *Sporting Mind Sporting Body: An Athlete's Guide to Mental Training,* Prentice-Hall, Englewood Cliffs, New Jersey, 1987.
12. S. Doheny-Farina, Writing in an Emerging Organization: An Ethnographic Study, *Written Communication, 3*:2, pp. 158–185, 1986.
13. B. Shneiderman, *Software Psychology: Human Factors in Computer and Information Systems,* Winthrop Publishers, Cambridge, Massachusetts, 1980.
14. B. W. Kernighan and P. J. Plauger, *The Elements of Programming Style,* McGraw-Hill, New York, 1978.
15. E. Yourdon, *Managing Structured Techniques: Strategies for Software Development in the 1990's,* 3rd Edition, Yourdon Press, New York, 1978.
16. W. L. Bryan and S. G. Siegel, *Software Product Assurance: Techniques for Reducing Software Risk,* Elsevier, New York, 1988.
17. M. W. Evans and J. Marciniak, *Software Quality Assurance and Management,* John Wiley and Sons, New York, 1987.
18. L. S. Levy, *Taming the Tiger: Software Engineering and Software Economics,* Springer-Verlag, New York, 1987.
19. R. A. Guillemette, Prototyping: An Alternative Method for Developing Documentation, *Technical Communication, 34*:3, pp. 135–141, 1987.
20. E. H. Weiss, *How to Write a Usable User Manual,* ISI Press, Philadelphia, 1985.
21. R. A. Grice, Using an Online Workbook to Produce Documentation, *Technical Communication, 30*:4, pp. 27–29, 1983.

22. K. Nichols and L. Duggan, Sharing Common Source Files for Documents: The Agony and the Ecstasy, *Proceedings* of the 35th International Technical Communication Conference, Philadelphia, May 1988, Society for Technical Communication, Washington, D.C., pp. ATA 137–140, 1988.
23. *Introducing Change Control*™, Context Corporation, Beaverton, Oregon, 1987.
24. M. Stefik, G. Foster, D. G. Bobrow, K. Kahn, S. Lanning, and L. Suchman, Beyond the Chalkboard: Computer Support for Collaboration and Problem Solving in Meetings, *Communications of the ACM, 30*:1, pp. 32–47, 1987.
25. M. Begemen, P. Cook, C. Ellis, M. Graf, G. Rein, and T. Smith, Project NICK: Meetings Augmentation and Analysis, *Proceedings* of the Conference on Computer-Supported Collaborative Work, Austin, Texas, December 1986, MCC Software Technology Program and the ACM, pp. 1–6, 1987.
26. S. R. Ahura, J. R. Ensor, and D. N. Horn, The Rapport Multimedia Conferencing System, *Proceedings* of the Conference on Office Information Systems, Palo Alto, California, March 1988, ACM and IEEE, pp. 1–8, 1988.
27. K. Lantz, An Experiment in Multimedia Conferencing, *Proceedings* of the Conference on Computer-Supported Collaborative Work, Austin, Texas, December 1986, MCC Software Technology Program and the ACM, pp. 267–275, 1987.
28. N. M. Delisle and M. D. Schwartz, Collaborative Writing with Hypertext, *IEEE Transactions on Professional Communication, 32*:3, pp. 183–188, 1989.
29. S. D. Conte, H. E. Dunsmore, and V. Y. Shen, *Software Engineering Metrics and Models,* Benjamin/Cummings, Menlo Park, California, 1986.
30. B. W. Boehm, *Software Engineering Economics,* Prentice-Hall, Englewood Cliffs, New Jersey, 1981.
31. P. S. Licker, *The Art of Managing Software Development People,* John Wiley and Sons, New York, 1987.