ISSUES IN INTEGRATIVE STUDIES
No. 15, pp. 125-147 (1997)

Co-Teaching Engineering and Writing: Learning about Programming, Teamwork, and Communication

by Louise Rehling, Ph.D.[1] and Lee Hollaar, Ph.D.[2]

[1]Technical and Professional Writing Program, San Francisco State University and [2]Computer Science Department, The University of Utah

Abstract: This study describes the rationales, method, and outcomes of a software engineering course that was co-taught by professors from the disciplines of computer science and writing. The course emphasized both teamwork and communication skills as critical to success for career programmers. The rationales for the course included theory and research in both computer science and writing. Additional rationales were identified in differences between industry practice and prevailing computer science course designs, plus differences between industry practice and student assumptions about programming professionalism.

The method of the course used an interdisciplinary process model grounded in Frederick P. Brooks' analysis of team programming practices. This model draws explicit links among teamwork, communication skills, and software engineering design practices. The outcomes of the course, reported by narration of representative learning experiences, included students' enhanced recognition of how technical writing and communication skills could improve programming performance. The authors recommend their interdisciplinary approach as a progressive course design for technology studies.

ALL OF THE MAJOR PROBLEMS associated with computer programming ... are ameliorated when programs and their dialogs with users become more literate.

—Turing Award winner Donald E. Knuth (1982, p. i).

Introduction

We were a bit worried at the start: What would students make of it? The crowning course required to complete their computer science major would be co-taught by teachers from the normally distanced fields of writing and computer science. Moreover, the course would be graded almost exclusively not on individual programming, but on reports and documentation for team-developed projects.

Fortunately, once they learned our rationales, students accepted our unusual course design. Throughout the course, we taught them how team programming and team writing could share a common process for addressing design and management issues. This approach helped them to derive lessons from their course experiences about how teamwork and communication skills could be critical to professional success.

We think that those were important learning outcomes and that our course could be a model for similar interdisciplinary courses in technology programs elsewhere. This report explains the rationales behind our course design, describes the approach to process that we taught, and illustrates the learning outcomes achieved.

Study Methods

We were both professors at the same four-year university—one of us from the Computer Science department in the College of Engineering, one of us from the Writing Program in the College of Humanities. During 1991-1993, we collaboratively taught two year-long sections of a required senior software seminar course.

Our course required students to design and complete a significant software development project—either real or realistic. Students had to work in teams. They also had to produce both a maintenance manual (including annotations of code) and a thorough, formally tested user manual. Students also had to develop written proposals, progress reports, and oral presentations. As teachers, we focused our in-class instruction and homework assignments on project planning, on selection of tools and design strategies, on effective teamwork, on usability testing techniques, and on skills and strategies for developing and executing the oral and written communication projects. Class discussions typically focused on student experiences with recommended processes and methods. We did not teach programming technologies, nor did we grade based on code features or program functionality.

Our courses served 79 computer science students, all seniors, who organized themselves into 43 different production teams: 31 of them involved individual students working with non-student fellow employees on the job, 6 of them involved multiple students working together with non-students on industry-sponsored projects, and 6 of them involved multiple students working on all-student, non-sponsored projects.

To gather data, we jointly interviewed each student in at least one 30-minute conference. We also conducted additional in-depth interviews with several students and student teams. In addition, we took notes on class discussions, presentations, and work products.

To validate our outcomes, we did confirming interviews with students. We also cross-checked our independent observations through joint discussion.

Course Rationale #1: Theory and Research in Computer Science

We knew, of course, the commonplace that computer programmers often work in teams to develop large software projects (see, for example, George Glaser, 1984; P Licker, 1985). Also, we knew that programmers often work in collaboration with technical writers, and, further, that effective collaboration between programmers and writers may be critical to product success (see, for example, Gold. 1989; Rivers, 1989).

Additionally, we recognized program annotation as a further link among programming, technical writing, and collaboration. As Turing Award winner Donald E. Knuth has explained, program annotation is a form of documentation writing that allows a programmer to collaborate asynchronously with other members of an extended and future programming team. That team consists of any and all who might, over time, work with an established code set to maintain it, adapt it, or develop product enhancements (1992). Program annotation is frequently an industry requirement, since businesses need to maintain and improve programs over an extended life-cycle. If code is not well annotated and the original programmers move on or up, expensive time and staff may have to be devoted just to understanding the original code base.

Knuth's notion of "literate programming" elevates program annotation to an expressive form, one that is central to programming if programs are to perform their "main purpose." Knuth argued that "the computer programs that are truly beautiful, useful, and profitable must be readable by people" (1992, p. I).

Knuth's methods, developed out of an entrenched "machine-oriented" coding establishment and promulgated over several years, perhaps were (and still may be) honored more in the breach than in the observance (Birss, 1992). However, Knuth laid the groundwork for new expectations of development teamwork and maintenance documentation (see, for example, Baecker & Marcus, 1990). These new expectations were further backed by cognitive studies that addressed program annotation (for example, Green, 1980). They were also buttressed by scholarship that addressed the linguistic relationships between artificial program languages and the grammar of natural human languages (Weizenbaum, 1966; Wright, 1978). The consensus seemed clear: shared communication issues are central to the work of programmers.

Also, even in 1991, hypermedia was already driving new conjunctions of technical communication and programming, by interconnecting graphical user interfaces with help screens, interactive customer service, and/or reference documentation. Separately, these elements could be conceived more simply as incidental to a program. However, computer science pioneer Ted Nelson (1987) noted that hypermedia offers possibilities that tend to overlap and even conflate these functions. Use of hypermedia, he pointed out, tends to blur distinctions between, on the one hand, inherent program qualities or functions and, on the other, program support. As a result, the concerns of programmers were becoming more and more those of writers, and vice-versa. Not surprisingly, Nelson's recognition was later echoed by a leading computer industry documentation expert, William Morton (1993).

Finally, industry practice had also changed the way technical writers were viewed and used. The trend was becoming to involve them throughout the product development cycle, from specification through user testing, as part of the product development team (see, for example, Bresko, 1991; Fowler & Roeger, 1986; Grice, 1983; Guillemette, 1987). This trend, of course, also involved programmers early and often in sharing the work of simplifying technical communication for users. This was a circumstance that their training often poorly prepared them for, since the field of software engineering is not only jargon-heavy, but jargon-proud as a form of professional identification and status (Barry, 1991). Nevertheless, if programmers were to work more closely with writers, programmers would need to value writers' work and be able to contribute to it cooperatively.

All of this literature from within the discipline of computer science demonstrated that its practitioners, and therefore our students, would likely more and more be expected to view programming practice differently. They would need to recognize it as intertwined both with teamwork and with writing projects and other forms of communication.

Course Rationale #2: Theory and Research In Writing

For several years before we developed our course, there had been a tremendous growth in scholarship concerned with collaboration in composition and technical communication. This was led by studies from Barbara Couture and Jone Rymer (1987) and from Andrea Lunsford and Lisa Ede (1991). Such studies both broadened our understanding of group writing and confirmed its prevalence in industry.

Cognitive studies of writing process also helped us to understand how professional identity might affect programmers' attitudes towards both writing and teamwork. Linda Flower (1987), for example, emphasized the complexity and criticality of task representation for writers. Composition researchers also recognized the importance of "the assignment students give themselves" (Greene, 1993) and suggested that more effective learning environments are context-bound (Ackerman, 1993; Russell, 1993). From this we saw how it could benefit computer science students to experience writing assignments not just in external courses, but integrated with their programming work.

This issue of learning context, scholarship informed us, becomes especially central when existing self-definitions may need to stretch. Students may be more receptive to forms of discourse from familiar communities (Harris, 1989) or from those with which they expect to identify professionally (Ronald, 1988). Dorothy A. Winsor's (1990) notion of how "engineering writing" and "writing engineering" twin, also explained how mastering writing forms appropriate to a technical profession may be definitional for professional achievement and identity. A similar process, we felt, would apply to our computer science students, who could benefit from having teamwork and communication assignments evaluated as measures of their professionalism. Later genre-based studies by Carol Berkenkotter and Thomas N. Huckin (1993) have further confirmed how disciplinary-specific professional writing can support self-definitions of professionalism.

Finally, ethnographic writing research also provided examples of how technical communication that is formally correct may nevertheless fail if it does not meet industry or professional norms (Paradis, Dobrin, Miller, 1989). This research, again supported by later case studies (Doheny-Farina, 1992), also suggested the importance in technical production teams of document-sharing as a tool for project management and of collectively developing writing expectations.

All of these studies from outside the discipline of computer science reinforced the professional value of teamwork and communication skills for technical experts working in industry. The writing scholarship also suggested that such skills could be more effectively taught via an interdisciplinary approach than by separate course work.

Course Rationale #3:
Industry Practice Versus Prevailing Course Designs

The computer science department in which we taught together has offered (then and now) excellent preparation in software engineering skills. However, perhaps like most computer science programs, ours offered courses that typically shared the characteristics described below—characteristics which differentiated students' course experiences from those they would be likely to have on the job in several key respects.

Short, one-shot projects

Development projects in term-long programming courses were typically small, discrete, test programs. They were not designed to be further developed and maintained, like programs in industry. As a result, such course projects did not require project planning, design reviews, and documentation.

Individual evaluation based on project functionality

Programming assignments—often submitted electronically and graded by teaching assistants—typically were evaluated based on accurate, appropriate applications of algorithms and efficient operations of complicated features. Because projects were designed to prove facility with programming languages and logic, not for use outside the classroom, these assignments usually granted little attention to usability features. Also, the absence of collaborative assignments, and, in some classes, the use of competitive grading curves, did not encourage teamwork.

Minimal writing requirements

Given that neither maintenance nor usability were typically issues in assignments or in grading, students often were not taught either program annotation or other forms of documentation in their computer science courses.

The computer science degree did require a technical writing course. However, the course was an introductory survey and not discipline-specific. It typically emphasized management writing (memos, reports, and so on), plus practical writing such as resume production. Lessons in rhetorical analysis, formats, and style often were not applied to the specialized kinds of technical documentation and communication that entry-level programmers would need to produce on the job.

Of course, many computer science courses may best be taught in the characteristic manner just described. However, in line with recent analysis (Wilson, 1996), we saw that teamwork and communication skills did not have the importance in our curriculum that they would have for students once they graduated and became programmers in industry. Therefore, we saw career-oriented grounds for differentiating the course format and expectations in our senior software seminar.

Course Rationale #4:
Industry Practice Versus Student Assumptions

Probably the strongest inspirations for our course design were our previous experiences in industry—one of us as a software engineer and project manager, one of us as a writer, editor, and communication manager. We had both learned on the job how central both teamwork and communication skills could be to programming success. Even more importantly, we had also observed that entry-level programmers often resisted that centrality, in ways that sometimes led to unnecessary frustrations for them and/or to expensive failures or delays for their companies.

Interestingly, excellent examples confirming our experiences came from several of our students at the start of our course. In the five stories that follow, students who were already working as workplace programmers experienced how good programming depended on good teamwork, good writing and communication, or both. Nevertheless, in all these examples, the students' closing "moral of story" quotations demonstrate how the students still disclaimed teamwork and communication skills as being part of their job description. These stories exemplify the need we saw: to provide a context for changing the entry-level attitudes represented here. (Throughout this report, given student names are pseudonyms.)

Bob E.'s story

Bob E. joined a work team that was programming a sophisticated robotic device. The device was feedback-sensitive, using real-time controls, so it performed interactively. It continuously measured the effects of its movements and correspondingly adapted those movements, based on the measurements. Consequently, the engineers programming the movements needed to work with the measurement programs; conversely, the engineers programming the measurements needed to work with the movement programs. Additionally, much of the programming work required inventing innovative and complicated algorithms. As a result, before Bob could write his program elements, he first had to locate, decipher, and interpret related program elements that others had already written.

Bob decided that good maintenance documentation would reduce the time that he would have to spend on that preliminary research. Therefore, he wrote a meta-program—a coding framework that had to be followed in order for other programmers' code to compile. This meta-program enforced maintenance documentation by requiring thorough program annotations and indexing.

Even though Bob's meta-program for maintenance documentation had to work first, before the robot could, Bob resented his documentation meta-program for taking time away from the "real work" of creating product programming. He saw the meta-program as a distraction, saying, "It wasn't my job to write it, but, unfortunately, I couldn't do my job well without it."

Todd W.'s story

Todd W. was assigned to adapt an application program that had been developed for use on Macintosh computers so that it instead could run on a Windows operating system. After writing his program adaptations, Todd provided his company's technical writer with a list translating the original set of Macintosh commands and executions into Windows equivalents. The writer used these to create a user manual designed for Windows users.

Todd, assigned to review the proofs as a quality control step, noted that the illustrations still represented Macintosh-style screens, instead of looking like the displays Windows users would see. He guessed that the disjunction would cause confusion among customers. However, Todd had only been told to check the text for accuracy. So, in his programmer's quality check he okayed without comment a manual that he predicted to fail. As a result, the user manual went to press without a change of illustrations.

As project programmer, Todd felt no responsibility for any consequent problems. Instead, he shrugged and said, "The technical writer didn't do his job and now customer service will pay the price."

Sam B.'s story

Sam B. was hired as programming manager at a credit union that used its own report management program. A programmer employee, who had written a substantial amount of the existing program, was annoyed at being passed over for promotion to Sam's job. As a result, he quit shortly after Sam was hired.

When Sam assigned a staff programmer to fix a bug in code that the ex-employee had written, the staff programmer could not understand the original code well enough to solve the problem. Neither could Sam. As a result, a large portion of the existing, unannotated code had to be thrown out—an expensive discard. Sam rewrote the program from scratch, a time-consuming process.

However, Sam accepted that development without criticizing the professionalism of the disgruntled ex-employee. Instead, Sam commented that "he probably was a good programmer, he just didn't write enough down."

Dee W.'s story

Dee W. worked for a company that developed medical systems software using a cooperative rapid prototyping development model. Working with only rough specifications, Dee's company delivered "first cut," untested, and partial programs to the customer. The customer then tried them out to develop additional requirements of what the system should do and how it should work. During that period, Dee and other programmers did all user training, talking the customer's testers through procedures on-site, without writing anything down.

Because of the many changes throughout the prototyping process, only after a prototype design was approved could technical writers begin developing a complete user manual for the system. Then they had only the initial product specifications and some sketchy notes from testers to work from. Since the cooperation involved for rapid prototyping did not extend to the documentation team, developing the manual took longer than originally expected. As result, Dee had to continue longer in an on-site training and support role.

Dee was frustrated, but he blamed the writers, saying, "They expect us [the programmers and the customer's testers] to do their job for them."

Dennis D.'s story

Dennis D. was employed by a small, innovative company that addressed a niche market with custom-tailored programs. Dennis and other programmers created brief, online help screens as part of the program development. If customers still had questions, there was no manual to which they could turn. However, they could directly query Dennis or other programmers for support. The programmers would then return messages to clarify operations or offer instructions.

Because the online helps did not contain all the reference information that customers typically needed, they found the system inefficient and frustrating. Also, Dennis responded to many repetitive queries, which was time-consuming. Using hypermedia for field service both decreased customer satisfaction and increased customer support time.

Nevertheless, Dennis still defended the system, saying, "At least we don't have to do documentation. So, instead of spending our time writing, we can just concentrate on being programmers."

Obviously, to all of these students, the definition of being a programmer did not include teamwork and writing or communication tasks, though those sometimes were recognized as necessary evils. The results of this dissociation were unfortunate in every case. We had ourselves seen the same kind of entry-level programmer dissociation when we were on the job, also with negative consequences. Therefore, we saw the need for a course that could create a context for learning different lessons from similar experiences. Upon graduation, we wanted our students to enter career positions recognizing, right from the start, that their teamwork and communication abilities would be as important aspects of their professionalism as their programming knowledge and skills.

Course Design: Our Interdisciplinary Process Approach

We agreed that the best way to join perspectives from our different disciplines would be by comparing team writing and team programming processes.

One of the classic texts on how to manage computer development projects is Frederick P. Brooks' *The Mythical Man-month* (1982). Its central analogy˜of software engineers mired in programming problems like prehistoric animals in a tar pit—problematizes the computer science field as being more than naive practitioners may have bargained for. Brooks offers a sophisticated perspective that is careful not to define programming professionalism atomistically. He insists that a "program" itself, although typically the individual programmer's own measure of productivity, is relatively useless. A program is suitable only for running by its own author on its own development system. Brooks, therefore, identifies the goal of programming more broadly than the production of programs. Moreover, he does so in ways that include both collaboration and technical communication as elements of purposeful programming.

Brooks notes that a useful "programming product" must serve multiple users in multiple environments, and, further, that it consists of testing, documentation, and maintenance, as well as the program itself. Since Brooks is concerned with huge systems programming, he further defines a "programming system" and an associated "programming systems product." The programming systems product is the ultimate goal of large systems projects. This goal requires even more, and more carefully managed, collaboration and technical communication throughout the programming effort (pp. 4-6). Brooks' text goes on to discuss several problematic design and management requirements for successful programming (in his larger sense).

Brooks sees collaboration as critical to the programming process and technical communication as a vital component of quality programming. As a result, we were able to draw from *The Mythical Man-month* several relevant approaches to design and management issues. We felt that these would apply to our students' teams in both the programming and the documentation tasks that we assigned to them.

What follows is our analysis of Brooks' process lessons for programmers in their connected roles as writers and team-players. We presented this analysis to our students. It gave us a way to bridge our disciplinary approaches and connect our assignments. It also helped students to recognize how their professional self definitions could include writing and teamwork (even though their previous academic studies in computer science had not included these roles).

As interdisciplinarity theorist Julie Thompson Klein has recognized, such an "organizing framework" can "[play] a vital role in the process of integration ... in interdisciplinary education" (1990, p. 191). Interestingly, our choice of Brooks' process model mirrored an approach that we encountered later: David Farkas chose Brooks' principles to illustrate the converse point for compositionists, of how writers could learn about teamwork from programmers (1990).

Brooks' design issues: conceptual integrity and over-design

Brooks defines conceptual integrity as the most critical of collaborative design issues for programmers. He notes that programming goals of complex, advanced functionality always must be balanced with ease of use requirements for simplicity and straightforwardness. These latter qualities can only proceed from a single-minded design specification that fully articulates how the program will work. However, when programmers work in peer teams, as in our course, schedule pressures encourage them to limit architectural time and control.

As Brooks points out, at the outset of a project, it can appear both more efficient and more appealingly democratic to define gross functions quickly. Team members can then parcel out among themselves responsibility for programming each of them. However, he warns that such pieced-together programs typically do not integrate the pieces well and completely. As a result, no matter how feature-rich, in the end. such programs confuse or frustrate users. The necessarily independent labor of implementing program modules, Brooks claims, must follow and continually refer back to an original and guiding concept for the whole.

On the other hand, Brooks also recognizes and warns against the danger that program architects may over-design. He discusses complexity as one of the seductions of the programmer's craft. As a result, he warns against the tendency of those who develop advanced software products to single-mindedly pursue embellishments at the expense of fundamental operational goals. As a counter to that tendency, he encourages "interactive discipline for the architect"—"early and continuous communication" with implementers. Such communication allows final design specifications to be cooperatively negotiated, so that they reflect programming realities (pp. 54-55). Self-discipline for architects is particularly important, Brooks notes, when they are subject to "second-system effect." This is the tendency to revisit an initial product design with a wish list that imbalances ease of use in favor of functionality (pp. 57-58).

Parallels with technical communication collaborations are clear. Brooks' requirement that a system "reflect a single philosophy" articulating "the specification as seen by the user" (p. 49) echoes classical rhetorical injunctions to write with a clear purpose and audience in mind. Collaborative writers can stray off that mark just as collaborative programmers can. As Wayne A. Losano notes (1985), a particular danger with multiple-authored technical manuscripts is leaving editing consistency as a late step in the composing process. At that point, separately created parts may not combine to create an integrated whole, unless an over-arching design informs each individually produced element.

We highlighted to our students how collaboratively producing user documentation presented the same kind of challenge for achieving conceptual integrity as they would face in their program designs. In fact. Brooks defines a program system architect as "the user's agent" whose job it is to plan "the union of the manuals the user must consult to do his entire job" (p. 2).

We also pointed out that, as a recursive process, writing documentation might, just like programming, require negotiated flexibility for avoiding over-design. Computer science students seem particularly inclined to edit with a desktop publisher. They are therefore susceptible to achieving slick-ness with minimal regard to clear organization and direct sentence-level style. Documentation writers, too, have their own form of "second-system effect": the addendum syndrome that can create complete, yet almost unusable, reference tomes.

Brooks' management issues: scheduling optimism and the Tower of Babel

Conceptual integrity also requires careful attention to management issues. The claim of Brooks' title, that the "man-month" is a "mythical" planning construct, derives from a representation of the software engineering process. A similar representation directly transfers to a model of the composing process for computer documentation writers. Programmers must work through design, code, debug, and prototype stages sequentially; similarly, manual producers correspondingly plan, draft, revise, and user edit. For large, team-produced projects of both types, the tasks are not perfectly partitionable but, to a large extent, are chained end-to-end. For example, programmers must have code in order to debug; manual writers must have draft copy in order to revise.

For groups working on both types of projects, like our student teams, sequential and interdependent scheduling becomes especially critical because accurate final manual production depends on final product. Further, what Brooks notes with regard to programming applies to writing as well: "the part that is easy to estimate" may not represent where most time will be spent (p. 20). Brooks cautions that programmers should not estimate an entire task counting just the time estimated for coding, which he gauges as typically only about one-sixth of a total programming project schedule (p. 20).

Likewise, it is a commonplace in composition instruction that as much time or more may be required for planning or editing as is required for drafting. Brooks notes that programming, as "an exceedingly tractable medium," encourages unjustified optimism about scheduling. Writing is similarly tractable. In team projects both programming and writing are susceptible to what Brooks dubs "regenerative schedule disaster" (pp. 21-22). This is an attempt to solve a delay by adding personnel to a task. The result may be to delay the task further, because of time needed for training and orientation by those who would otherwise continue producing.

Optimism about scheduling is abetted by problems of poor communication and organization that Brooks discusses by way of an analogy to the Tower of Babel failure. Brooks identifies ongoing technical communication (both via documentation and via informal and formal meetings and presentations) as critical to maintaining interactive reviews. Such reviews can keep both conceptual integrity and scheduling on track. He also suggests that team roles reflect a willingness to define "democracy" broadly, by recognizing the need for structured interactions. He recommends a distinction of design, implementation, and documentation task assignments in a "surgical team" model for small groups, like the ones in our class.

This recommendation, too, has parallels in group writing practices that, in workplace writing, typically require less joint drafting than shared planning and editing (Couture & Rymer, 1989). Documentation, too, we pointed out to students, cannot spin off independently of other tasks; teams need to maintain consensus and clarification for all their interrelated project assignments. Implementing Brooks' focus on team management, of course, required our students to focus on collaborative planning and communication skills. This meant breaking away from the individual control that they were accustomed to in previous course work.

Course Learning Outcomes

There was value in drawing parallels between programming teamwork and writing teamwork. There was also value in recognizing the professional practice of disassociating programming professionalism from documentation. These emphases encouraged students to analyze their project work in new ways. We hoped that they would recognize the synergisms of combining effective teamwork and writing (or other communication) with programming. We wanted our students to see both teamwork and documentation as essential and transformative elements of their computer science practice.

Of course, maintaining Brooks' professional standards was a difficult challenge to meet, both for our students' programming projects and for their associated maintenance and user documentation. Both as programmers and as manual drafters, the students needed to strive for conceptual integrity, first, by establishing clear design parameters, then, by usefully revisiting those, based on discoveries or problems encountered in execution. The students also needed to manage documentation production in a parallel and coordinated way with programming production, using a collaborative structure and communication system that worked for both endeavors.

Because the process requirements were so difficult, and so novel for most students, teams experienced mixed results with putting the course's professional principles in practice. Although most project outcomes were positive, the range among all teams spanned the success-failure continuum.

However, all outcomes were consistent in another, more important way: for every team in our course, effective documentation and teamwork were the make-or-break elements for project success. High quality programs simply did not happen without good writing and teamwork strategies.

To illustrate this relationship using the crudest measure, none of the students' project teams produced "C"-level documentation and "A"-level programs, or vice-versa. Instead, the success of projects depended on positive synergism among all three elements: programming, teamwork, and writing. More specifically, in the most successful projects, each member of the team was effective in each assigned role—as programmer, team-player, and/or writer. When any team member neglected or performed poorly in any assigned role, the entire project suffered, including the programming product.

Most importantly, since we required final reports analyzing team performance using our interdisciplinary process framework, members of every team recognized the make-or-break relationship just described. That created, of course, the opportunity for less-than-successful projects nevertheless to be positive steps toward professional growth.

We chose the three stories below because they illustrate different degrees of project success or failure.These post-course stories can be usefully contrasted with the initial student stories told above (within our discussion of student assumptions versus programming practice). In those earlier stories, students drew "moral of the story" lessons that dissociated teamwork and communication skills from programming professionalism.

In contrast, in the following post-course stories, students draw more sophisticated understandings from their project experiences.

The game design team's story

Some teams failed to achieve conceptual integrity in both programming and writing arenas. Perhaps the most telling example of this was a four-member student team that set out to program a development tool for designers of board games.

The game design team's individual members were highly regarded, competent programmers, each of whom had performed exceptionally well in previous computer science course work. But the students in this team ended the course in self-acknowledged defeat. They were unable to demonstrate successfully even the simplest level of program application for designing any basic board game. In addition, their user documentation provided an unusable instruction set that was critically incomplete and disjointed. Finally, their maintenance documentation was limited to brief code annotations that they could not easily share.

These students worked hard: each one programmed a module, each annotated that module, and each drafted an associated manual section that displayed wit and formatting sophistication. However, the team product could not demonstrate the power of the students' individual ideas. For example, the team forecast grandiose interactive capabilities, but no one designed in even basic, critical variables for board play, such as non-linear piece movement. The necessary scope of their project became clear to these students only near the end of the term, when sufficient time no longer remained to realize key objectives.

Members of this team offered explanations for their disappointing results that had everything to do with the absence of conceptual integrity.They realized that their notions of overall product use had remained fuzzy and unrelated. Meanwhile, they had over-designed, dreaming up "bells and whistles" features. One student, Ari J., concluded on behalf of the team that "we had some good parts, but they didn't add up to a whole."

Ari also noted that he and his teammates had originally been proud to see themselves as "programming gunslingers" whose individual brilliance and due date heroics would always pull them through. However, in the end, these students were no longer proud of that self-definition: they saw how it had led to scheduling optimism and Tower of Babel-style communication problems. They now anticipated taking design and management approaches to on-the-job projects that would be quite different from the approaches that had worked so well for them in previous course work.

The product support team's story

The most successful teams made provisions for conceptual integrity in developing both their programs and their documentation. In these cases, fruitful exchanges took place between the collaborative programming process and the writing process. By recognizing and addressing conceptual integrity as a common issue both for their programming tasks and for their user documentation tasks, they made use of writing to accomplish both professional goals. Written guidelines—specifications or other design plans, annotation formats, and even user documentation—were useful checks against over-design. Attempting to explain a programmer's computing procedure in operational terms sometimes convinced a team of the feature's relative inutility, given the difficulty of use.

In the following example, a combination of such synergistic advantages is illustrated. Documentation was not an add-on to the programming project, but its core articulation and the means for project management.

Matt P. was employed as the manager of a programming team charged with developing an internal product support project for a large computer engineering firm. Matt's supervisor had told him that no user documentation would be required for his team's support product, since only technical personnel (experienced programmers and customer service representatives) would need to use it. However, because our course required a user manual, Matt wrote one, modified his design somewhat accordingly, then provided his draft to his programmers as an advance design guide, a la Brooks.

The results impressed Matt and also earned him a commendation from his supervisor, for two reasons.

First, the program was developed faster and with less hands-on programming involvement by Matt than had been required for previous team developments. His programmers, Matt reported, jotted down implementation notes for each of his user instruction steps right in their copies of his manual. Those notes were used to discuss architectural issues interactively. Then, the notes were efficiently assembled to create maintenance documentation. The programmers' modules pieced together without requiring the kind of significant revising that Matt had done in the past on similar projects.

Second, after the product was programmed, technical personnel used the user manual after all. Consequently, they made more reliable use of this support product than they had of others—enough so, that it was used as the base for future product support enhancements. Of course this use was facilitated by the readily available maintenance documentation.

"Without the manuals," Matt commented, "the product would have cost more and been worth less to the company. It really wouldn't have been the same product."

Although Matt had started our course with on-the-job programming experience, this project was his first attempt to conjoin principles of teamwork, writing, and software engineering. He internalized Brooks' central tenet that effective practice involves making collaboration and communication central to programming. He also experienced the professional benefits of re-defining his own professional practice through these means.

The genetic database team's story

Interestingly, some students who were cognizant of risks to conceptual integrity in program design, and who undertook steps to control those risks, did not take a similar approach to documentation. In the end, both programming and documentation tasks typically suffered from that disjunction. The following example illustrates that result. It tells the story of a student team in which two independent students joined an employed student to work on her industry project.

Elyse R., the student employee, saw her major challenge as bringing the outsider students up-to-speed on the scientific knowledge required for her company's programming project. The task was to update and expand an automated recording and classifying system for genetic identification data. She firmly took control of programming architecture, spending considerable time and effort talking to others in the team, describing intended system operations. She then assigned the other students programming modules to implement and annotate, asking them also to write user documentation for them.

Progress, Elyse reported, was much slower than expected, for two reasons.

First, in designing a program enhancement, Elyse had fallen prey to second-system effect over-design. The student programmers attempted to write code to her new requirements. However, without written maintenance documentation for her previous work, when problems occurred interactive communication proved time-consuming.

Second, Elyse also had to spend considerable time rewriting and reformatting the others' manual sections. Elyse had organized the original product manual procedurally, providing a step-by-step, tutorial-like instruction set. However, the other students organized their write-ups around Elyse's briefing topics: they produced essay-style text describing program functions and outputs, plus scientific backgrounding which Elyse's technical users already well knew. Also, Elyse lacked an established maintenance documentation framework or annotation standard. Therefore, working on the others' manual sections and trying to understand their coding problems limited the time she could spend designing and coding the program itself.

As a result, Elyse concluded, "None of us got the amount of project work done that we'd intended to do."

Elyse and her student teammates ruefully acknowledged what they had refused to recognize at the start: that their project work could not neatly divide into separate programming and documentation tasks. Initially positioning technical communication in a secondary, support role hid the potential value of documentation for improving programming itself.

All of the students' evaluative comments at course-end demonstrate that they came to recognize how conceptual integrity required integration of documentation and programming practice. This was true not just because documentation was necessary, but because il could define and transform product design, direct code implementations to take unanticipated forms, and address future as well as current needs. In the words of a student who (using Brooks' surgical team model) was assigned the role of chief documenter, "I created the program as much as anyone who wrote code." This student's experience, along with the examples above, also illustrates how good teamwork—managing documentation production in a parallel and coordinated way with programming production—was critical to achieving conceptual integrity.

Students like Matt, who integrated the management of programming and documentation tasks, applied conceptual integrity consciously and early on. He and his team used documentation to achieve larger programming goals, taking the same interactive approach to team writing documentation as they took to team programming. Although, they were sometimes drafters, sometimes coders, they used recursive, interactive planning and revising in both roles.

On the other hand, students like those in the game design team, whose practice of programming was pre-professional, worked as a team only in name. As a result, both their programming and their documentation failed (even in their own estimates) to meet professional criteria. The game team students' final recognition of the value of true and parallel collaboration in both endeavors challenged the hacker mentality that they had brought to the course.

Most telling were students like Elyse and the others in her genetic database team. They recognized conceptual integrity as a professional value, but isolated it from technical communication. Because team members took a different approach to documentation than to programming issues, their overall professional accomplishments were limited. These students ended up, however, understanding how their disjointed approach flawed their work and productivity. The model from Brooks provided by our interdisciplinary process method helped our students, as Klein recommends, to "deal openly with the idea of integration" in order to attain "a heightened sense of the meaning of interdisciplinary work" (1990, p. 191).

Implications

Our students' learning outcomes are what validate the course rationales and design for us. We gave our students experiences with teamwork and communication assignments that they might well face on the job. Additionally, we taught them an approach for using teamwork and communication to develop their professionalism as software engineers. Not incidentally, that included developing new respect for technical writers who might be part of a software product development team. Co-teaching made these outcomes possible, because we each needed to contribute disciplinary expertise and teaching experience to cover the course's range of topics and assigments. We had, as James R. Davis notes team teachers of interdisciplinary courses are bound to do, "invented a new subject" (1995, pp. 47-52).

We must note, nevertheless, that our success was institutionally problematic. Although we recommend our course as a model for similar interdisciplinary collaborations, there is a lesson, too, in its programmatic aftermath. Our jointly designed course continued to be taught successfully, with different co-teaching writing instructors, for an additional two years, 1993-1995. After that, though, when the co-teaching computer science professor moved to another course assignment, the senior software seminar was re-designed. Our "new subject," including our course's emphasis on teamwork and communication skills, was not yet an established objective for the program's curriculum, nor, perhaps, for computer science education in general.

However, that day of inclusion may come. A recent Chair of the Computer Science Accreditation Committee has observed that "academic institutions are not compelled to rethink and reshape their computer science curricula." Further, he identified both teamwork and communications skills as being critical needs. He also called for appropriate curricular "experimentation and innovation" (Wilson, 1996, pp. 54, 59).

These goals doubtless have echos in other engineering and technology programs. Many graduate technically trained students who will require teamwork and communication skills when they apply subject matter knowledge to real world projects. Interdisciplinary courses such as ours could be part of a new movement to revise education for technical professionals by using integrative frameworks for teaching and learning.

Biographical Note: Louise Renting directs the interdisciplinary Technical and Professional Writing Program at San Francisco State University, where she is an Assistant Professor. Before returning to college teaching several years ago. Dr. Rehling worked in industry for over 15 years, as a writer, editor, manager, trainer, and communication consultant. Her Ph.D. is in English, from the University of Michigan.

Lee Hollaar is a Professor of Computer Science at the University of Utah, where he has taught computer hardware and software design and currently teaches intellectual property and computer law. On a recent sabbatical leave in Washington he served as a fellow with the Senate Judiciary Committee and a scholar-in-residence at George Washington University. His Ph.D. is in Computer Science, from the University of Illinois at Urbana-Champaign.

References

Ackerman, J. (1993). The promise of writing to learn. *Written communication*, *10* (3), 334-70.

Baecker, R.M. and Marcus, A. (1990). *Human factors and typography for more readable programs.* Reading, MA: Addison-Wesley.

Barry, J.A. (1991). *Technobabble*. Cambridge, MA: MIT Press.

Berkenkotter, C. and Huckin, T.N. (1993). Rethinking genre from a sociocognitive perspective. *Written Communication*, *10* (4), 475-509.

Birss, R.C. (1992, September). Literate programming [review of the book Literate programming]. *Computer*, *25* (9), 126-127.

Bresko, L.L. (1991). The need for technical communicators on the software development team. *Technical Communication*, *38* (2), 214-220.

Brooks, F.P. (1982). *The mythical man-month: Essays in software engineering*. Reading, MA: Addison-Wesley.

Couture, B. and Rymer, J. (1989). Interactive writing on the job: definitions and implications of "collaboration." In M. Kogen (Ed.), *Writing in the business professions* (pp. 73-93). Urbana, IL: National Council of Teachers of English.

Davis, J.R. (1995). *Interdisciplinary courses and team teaching: New arrangements for learning.* Phoenix: American Council on Education and Oryx Press.

Doheny-Farina, S. (1992) *Rhetoric, innovation, technology: Case studies of technical communication in technology transfers*. Cambridge, MA: MIT Press.

Farkas, D.K. (1990). Collaborative writing, software development, and the universe of collaborative activity. In M.M. Lay, and W.M. Karis (Eds.), *Collaborative writing in industry: Investigations in theory and practice* (pp. 13 -30). Amityville, NY: Baywood Pub.

Flower, L. (1987). *The role of task representation in reading-to-write*. Berkeley, CA: U. of California. Pittsburgh, PA: Carnegie Mellon U.

Fowler, S. and Roeger, D. (1986). Programmer and writer collaboration: Making user manuals that work. *IEEE Transactions on Professional Communication*, *29* (1), 21-25.

Glaser, G. (1984, October). Managing projects in the computer industry. *Computer*, *77* (10), 45-53.

Gold, E. (1989). Bridging the gap: In which the author, an English major, recounts his travels in the land of the techies. In C.B. Matalene (Ed.), *Worlds of writing: Teaching and learning in discourse communities of work* (pp. 335-342). New York: Random House.

Green T.R.G. (1980). Programming as a cognitive activity. In H.T. Smith and T.R.G. Green (Eds.), *Human interaction with computers* (pp. 271-320). New York: Academic Press.

Greene, S. (1993). The role of task in the development of academic thinking through reading and writing in a college history course. *Research in the Teaching of English*, *27* (1), 46-75.

Grice, R.A. (1983). Using an online workbook to produce documentation. *Technical Communication, 30* (4), 27-29.

Guillemette, R.A. (1987). Prototyping: an alternate method for developing documentation. *Technical Communication, 34* (3), 135-141.

Harris, Joseph (1989). The idea of community in the study of writing. *College Composition and Communication, 40* (1), 11-22.

Horton, W. (1993). Let's do away with manuals ... before they do away with us. *Technical Communication, 40* (1), 26-34.

Klein, J.T. (1990). *Interdisciplinarity: History, theory, and practice*. Detroit: Wayne State U. Press.

Knuth, D.E. (1992). *Literate programming.* Chicago: U. of Chicago Press.

Licker, P.S. (1985). *The art of managing software development people*. New York: Wiley.

Losano, W.A. (1985). Editing for style and consistency: The multiple author manuscript. In CD. Rude (Ed.), *Teaching technical editing* (pp. 63-71). N.P.: Association of Teachers of Technical Writing.

Lunsford, A. and Ede, L. (1986). *Singular texts/plural authors: perspectives on collaborative writing*. Carbondale, IL: Southern Illlinois U. Press.

McKay, L. (1984). *Soft words, hard words: A common-sense guide to creative documentation*. Culver City, CA: Ashton-Tate.

Nelson, T. (1987) *Literary Machines: the report on, and of, Project Xanadu concerning word processing, electronic publishing, hypertext, thinkertoys, tomorrow's intellectual revolution, and certain other topics including... Ed. 87.1.* Swarthmore, PA: Theodor H. Nelson.

Paradis, J.. Dobrin, D. and Miller, R. (1985). Writing at Exxon ITD: Notes on the writing environment of an R&D organization. In L. Odell and D. Goswami (Eds.), *Writing in nonacademic settings* (pp. 281-307). New York: Guilford.

Rivers, W.E. (1989). From the garret to the fishbowl: thoughts on the transition from literary to technical writing. In C.B. Matalene (Ed.), *Teaching and learning in discourse communities of work* (pp. 64-79). New York: Random House.

Ronald, K. (1988). On the outside looking in: Students' analyses of professional discourse communities. *Rhetoric Revew, 8* (l), 130-149.

Russell, D.R. (1993). Vygolsky, Dewey, and externalism: Beyond the student/discipline dichotomy. *Journal of Advanced Composition, 13* (1), 173-197.

Weizenbaum, J. (1966). Eliza—a computer program for the study of natural language communication between man and machine. *Communications of the ACM*, *9* (1),36-45.

Wilson, Kenneth E. (1996, January). The nature of a good and accredilable computer science program. *Syllabus, 9* (4), 54, 59.

Winsor, D.A. (1990). Engineering writing/writing engineering. *College Composition & Communication, 41* (1), 58-70.

Wright, P. (1978). Feeding the information eaters: Suggestions on integrating pure and applied research on language comprehension. *Instructional Science, 7* (3), 249-312.