Playpens for Mind Children: Continuities in the Practice of Programming

Patricia Galloway

Like most activities, the practice of computer programming involves real people trying to get their work done, and it has a complicated, if relatively short, history in its modern manifestation. This article addresses some of the early computer science discussions of programming and theories about how it should proceed. The article closes with a discussion of the more recent turn to so-called agile methods, demonstrating that some of the problems and practices of computer programming demonstrate a remarkable continuity over forty years in spite of much-promoted new approaches and changes in the computing environment.

Programming computers is by far the hardest intellectual task that human beings have ever tried to do. Ever.

—Gerald M. Weinberg, *Understanding the Professional Programmer*

A good process is organic, embodied in the habits and conversations of the team. Like any behavior, you can document it, you can do your best to guide its development, but attempts to enforce it by strict mandate are more likely to encourage rebellion than participation.

—Marc Rettig and Gary Simons, "A Project Planning and Development Process for Small Teams"

In our present information society, computer programmers and their practice are at the heart of the infrastructure that supports and constrains nearly all the information activities in which we are involved and engaged. That infrastructure is increasingly hidden because most people accept it as such, and yet, as legal scholar Lawrence Lessig observed, in the world of cyberspace it is computer code that is law.[1] For this reason we should be interested in programmers, their practice, and the history of that practice. It is taken for granted that programmers are engineers or scientists, that they are always mostly in control of the work they do, and that they are careful to make that work "correct" in some sense. But it is not in fact clear what the status of programmers is, even to them: that is, whether they are part of a science or a profession, and

Information & Culture, Vol. 47, No. 1, 2012
©2012 by the University of Texas Press, PO Box 7819, Austin, TX 78713-7819

whether they share an ethic that will somehow guarantee the infrastructure that we must perforce trust them to build.

The historical emergence of programming as a profession has been documented.[2] This article pursues a more modest goal, as it is interested in a microhistorical fragment of programmers' experience of carrying out the process of programming, particularly as it is most often now done in teams. It is in this sociotechnical context that the program-artifact is made, and ultimately the precise way it is made is strongly influenced by the interactions of the members of programming teams, by the technicalities of programming languages and digital programming environments, and by the relationship between the programmer and the program as creative product.

To explore the question of the individual role in the creation of a program product, I want to place a particular case in its historical context as framed through research in the computer science literature on software engineering and software design. The case is the work of students in an emblematic and venerable software engineering (SE) course, designed by one of the creators of the SE concept, at a historical moment in the early 1990s when SE as an instrument for the control of programming practice was beginning to be replaced by new ideas brought in by the growing dominance of personal computers and networking. The ethnographic observations documenting the programming practice in the course, collected by participant observation and including formal interviews and documentation generated by the programming team, were designed to observe an entire programming project from start to finish in order to understand how programmers made programs "situated in cultural systems, social relations, and institutional matrices," with the goal of helping to design a fully digital environment to support programming work.[3] I have used these observations to explore the experiential underpinnings of the creation of a community of practice, how it developed through the construction of a local work practice, how the students' own situations in the relationships the project generated influenced their work, and what agency individual programmers enjoyed in using their creativity to put their own stamp on the resulting product. In this historical moment, we can see as under a microscope the beginnings of the emergence of the kind of team practice that is widespread today in the production of the kinds of software that I am using to write this article and you may be using to read it. Like all historical process, it needs to be seen in closely observed moments, and I think this case allows us to see an important moment as well as its evolutionary connection with what went before and what has come after.

Programming as "Discipline"

More than thirty years ago, in a critical monographic study of the social context of computer programming, sociologist Philip Kraft argued that the practice of computer programming was in the process of going the way of its victims, industrial workers who had been pushed out and deskilled by automation: "Programmers, systems analysts, and other software workers are experiencing efforts to break down, simplify, routinize, and standardize their own work so that it, too, can be done by machines instead of people."[4] Kraft based his argument on the work of Harry Braverman and David Noble.[5] He used interview techniques to gather data on the basis of which he classified software workers, from least to most skilled, into coders, programmers, and analysts, arguing that management had used canned programs, structured programming, and "Chief Programmer Teams"—all techniques falling under the rubric of SE—to reinforce and reproduce the deskilling process, although he admitted that by 1977 the process was not complete because of the intellectual nature of the work.[6] Two years later Kraft repeated the claim in an article summarizing the earlier work, although he then distinguished "high-level programming languages," rather than Chief Programmer Teams, as the third of the routinization techniques.[7]

Kraft was rather too easily persuaded in taking as ominous warning what we can now see as managers' overly optimistic forecasts of complete automatization of the programming production process.[8] More recently, Bruce Berman has similarly erred in simply adopting Kraft's thesis and updating it slightly by adding a diatribe against the pretensions of artificial intelligence research.[9] Since the 1970s structured programming in "high-level" or third-generation languages, demonized by Kraft, has in its turn been superseded by further innovations in programming technology and practice (object-oriented programming [OOP], or agile programming), yet the managerial nirvana of programmerless computing has not been realized—indeed, development has moved in the opposite direction.[10]

These alleged leanings toward routinization are part of a historical process involving the production of hardware and software in the context of business organizations, educational institutions, and a political economy never far removed from the potential military and governmental applications of these technologies. What is most important here, I would suggest, is that in the course of the development of hardware devices and the software that performs information manipulation supported by them, the software has proved easier to adapt than the hardware, such that with time software layers have been added to permit greater abstraction in the programming process and indeed in the use of computers in general; the greater the abstraction, the harder it is for nonprogrammers to understand what is going on. Over time, programmers have created all these layers, and continue to do so, but the so-called lower-level work of writing operating systems and programming language compilers as well as designing graphical user interfaces is done less often. This is because in a real sense it is infrastructural, whereas so-called applications have become increasingly diversified and commodified and include not only programs that assist in work but also those that can provide for almost any other informational (and creative) desire. And since all these layers remain with all their combined complexity, and mastery of them in greater or lesser degree is required for programmers, it has become increasingly difficult to routinize their work.

Neither Kraft nor anyone else could have forecast that computers would multiply as they have. When Kraft originally wrote, the number of mainframe computers in the world could be counted in the hundreds, minicomputers were still new, and microcomputers had just appeared on the market in do-it-yourself kits. Today, while the number of mainframe computers has not significantly grown, the capabilities of minicomputers are now surpassed by the millions of microcomputers that are ubiquitous in business and in homes, while portable devices more powerful than many personal computers—enabled by wireless technologies and Internet connections—are now becoming ubiquitous as well. In many environments, the large central mainframe computer has also become a thing of the past, giving way to networks of personal computers on every desk as the model of reticulated managerial hierarchies has collapsed into drastically flattened corporate networks and powerful centralized computing resources are shared via the Internet. The Bureau of Labor Statistics' *Occupational Outlook Handbook, 2010–2011 Edition* forecasts that the occupation of "computer software engineers and computer programmers" will experience 21 percent growth, adding 283,000 additional jobs for a total of 1,619,300 by the end of the period 2008–18. Nor will these jobs be badly paid or poorly recognized, for salaries in the field have steadily climbed since 1977.[11] Certainly, many of the managers have become expendable; but have the programmers internalized a discipline?

There can be no argument with the claim that a division of labor that creates autocratic managers and lowly programmers as well as dominance-ridden hierarchies of control empties the programming task of its joy and creativity.[12] Nor has the conceptualization of computer programming as SE, which Kraft argued was meant to bring it under discipline, gone away. Invented as a solution to the "software crisis" of the 1960s (due to a drastic shortage of programmers), SE was indeed modeled, as Kraft suggests, on the electrical (hardware) engineering process. But its adoption as an ideology of control was not particularly successful for management; huge gains in programming efficiency were not realized and the software crisis continued. As a popular synthesis indicates, the faith of computer scientists in the efficacy of the original concepts of SE for taming the complexity they faced was due to their naïveté in believing that software could be engineered like computer hardware, when in fact it differs in fundamental ways: (1) software does not require material manufacture, as it can be changed with relative ease at any time; (2) software is evaluated by judgment and intuition; (3) software has no practical bound on complexity; and (4) software does not wear out, so its reliability depends on the number of errors it contains to begin with.[13] Software, in other words, can be fully defined before it is written about as well as a novel can. Just because the original notions of SE, with its "waterfall model" of step-by-step sequential development and its detailed management structures, were so little cognizant of the actual character of the programming task as socially situated intellectual work, the kind of hard-and-fast implementations of such schemes as Kraft criticized have turned out to be self-limiting, and students being taught SE are often advised by their instructor to circumvent them: "The waterfall model is a wonderful way to explain what happens, but it is not a wonderful way to build a system. It is a wonderful way to organize things ex post facto as though in an ideal world it had happened that way."[14]

Thus after nearly fifty years of SE, large programming projects still fail to meet deadlines and deliver promised features, and the consequences of software failure (i.e., Internet-wide computer-virus attacks, collapse of the entire AT&T long-distance telephone system, failure of onboard space shuttle computer systems, and cyberwar attacks on companies and small countries) have become more serious in terms not only of potential loss of profits but of loss of life. Managers still lament the same problems of managing programming teams, even after five decades of ingenuity have been applied to the creation of ever more supportive and friendly programming environments, numerous evangelistically promoted approaches to software design, increasingly micromanaged software design schemes, and, most recently, agile programming models that hand over most responsibilities to the programming team itself but encourage adherence to a strict framework. This should not be surprising in view of the increased complexity of software, which is itself due to an increase in the demand for an accessible, "user-friendly" computer environment that allows computers to be placed under the direct control of workers.

The intellectual work of computer programming therefore remains intractable to routinized control.[15] Steady innovation in the field of computing makes every programming effort a new creation and, more significantly, still leaves programmers in control of a scarce and valuable skill. Where commercial software for any computer is concerned, although the program becomes a prototype that is subsequently commodified, its original creation remains intellectual craft work, and upgrading and addition of new features demand more of the same. With the explosion of the market for microcomputer software, fierce competition for programming talent has led to the building of corporate facilities for programmers that resemble luxurious college campuses more than regimented factory floors, and the kind of creative idiosyncrasy that characterizes the world of so-called hackers in academic computer centers is not only tolerated in these commercial environments but, some would say, encouraged.[16] The point is that programmers on the whole are not in serious danger of their work being deadened and restrained by routine; indeed, it is clear that they are still envied by other intellectual workers. Citing a project in which engineers from his company—not the typically deskilled engineers discussed by Kraft but elite computer network engineers—collaborated with the pampered programmers from a software company, one manager said: "Our engineers were forced to suffer daily indignities in the face of these obnoxiously arrogant programmers. One of my friends referred to them as the Hitler Youth."[17] But what, then, is so special about programming work? What insulates programmers and even guarantees them special privileges? And has the situation changed over time?

Programming as Intellectual Work: Two Historical Views

Recognition of the special character of the intellectual work of programming has never been lacking among academic programmers and software designers themselves. Perhaps the most often quoted book on software engineering, written by one of the inventors of the field, is Frederick Brooks's 1975 book of essays, *The Mythical Man-Month.*[18] This book is still used and referred to because it constitutes a cogent and credible description of practice developed from actual experience of large design projects, including one of the first widely successful commercial mainframe computers, IBM System 360. Brooks was the first to argue influentially for the idea that software teams need to be small and autocratically managed by a creative design genius to achieve the conceptual integrity that he saw as the only saving grace in the face of nearly uncontrollable complexity. The book's title refers to his recognition that a central problem of team programming is communication among programmers, a difficulty that increases as teams are enlarged. He laid down standards for team structure, software documentation, and the principle of planning trial prototypes for discard, and much of his thinking has continued to have relevance because he realized early that software production was always likely to be affected by uncertainty. His description of the "joys of the craft" contains a classic passage that links the special nature of programming to the literary activity that has become synonymous with creativity for modern Western culture:

> The programmer, like the poet, works only slightly removed from pure thought-stuff. He builds his castles in the air, from air, creating by exertion of the imagination. Few media of creation are so flexible, so easy to polish and rework, so readily capable of realizing grand conceptual structures. . . . Yet the program construct, unlike the poet's words, is real in the sense that it moves and works, producing visible outputs separate from the construct itself. . . . The magic of myth and legend has come true in our time.[19]

Another theme here offers an important key to the unique nature of programming: the program itself is a form of practice. Brooks even alludes to magic, the magic of a sorcerer's apprentice. (It is no accident that expert programmers are referred to as sorcerers and wizards.)[20] Brooks's wizards are the elite of the programming world, however; throughout the book his vision is phrased in terms of the single expert programmer, and he always emphasizes the need for what he calls "uno animo" at work in the overall design of the software project.

In 1986 Brooks updated that vision in an influential article, "No Silver Bullet," expanding on the imagery of computer programming sorcery, this time mobilized against software project monsters, werewolves that "transform unexpectedly from the familiar into horrors."[21] Hollywood images of werewolves pursuing an attractive young woman, her hair streaming and her gaze cast apprehensively over her shoulder as she flees, were even used by the editors of the IEEE *Computer* magazine, both for their cover art and in the text of Brooks's essay itself, when they reprinted it from the sober conference proceedings in which it first appeared.[22] Brooks discussed the fundamental reasons why "magic bullets" of the past—high-level languages, time-sharing, unified programming environments (including many of the demons Kraft listed)—and further proposed wonderful schemes—high-level language advances, object-oriented programming, artificial intelligence, expert systems, "automatic" programming, graphical programming, program verification, new environments and tools, and workstations—had not and would not provide order-of-magnitude improvements in programmer productivity. The key, he argued, lay in the fact that they all addressed the "accidents" of the activity of computer programming and did nothing to attack its "essence," which he saw as consisting of four elements:

1. Complexity: "Software entities are more complex for their size than perhaps any other human construct because no two parts are alike. . . . In most cases the elements interact with each other in some nonlinear fashion, and the complexity of the whole increases much more than linearly."
2. Conformity: "Much of the complexity . . . is arbitrary complexity, forced without rhyme or reason by the many human institutions and systems to which [the programmer's] interfaces must conform."
3. Changeability: "The software product is embedded in a cultural matrix of applications, users, laws, and machine vehicles. These all change continually, and their changes inexorably force change upon the software product."
4. Invisibility: "The reality of software is not inherently embedded in space. . . . [W]e find it to constitute not one, but several, general directed graphs superimposed one upon another. . . . This lack not only impedes the process of design within one mind, it severely hinders communication among minds."[23]

The computer programmer was here seen as subject to the irrational requirements of "human institutions" and change in the cultural environment of computing. Brooks's prescription to solve these problems had four parts, some of which echo Kraft's concerns, some of which point in a different direction:

1. "Buy, don't build": A mass market for software would lead to the possibility of making major economies.
2. Requirements refinement and rapid prototyping: Focus on communicating more effectively with users to understand their requirements and offer them working prototypes for criticism and usability testing before proceeding to full-scale development.

3. "Grow, don't build": Brooks heartily recommended incremental development through getting small parts of a system running early, then "growing" the rest of the system around them, stating that "enthusiasm jumps when there is a running system, even a simple one."
4. Great designers: "Whereas the difference between poor conceptual designs and good ones may lie in the soundness of design method, the difference between good designs and great ones surely does not. Great designs come from great designers. Software construction is a creative process."[24]

This last item, the great designers, echoes Brooks's earlier emphasis on the notion of coherent, one-mind design for large software projects; it is still a notion that depends upon powerful solo designers with great authority in enforcing their vision. It is a notion that is also reflected in Brooks's vision of the development of computer science through "driving problems" adopted from other fields.[25] But we also see here several crucial insights about the need for both programmers and users of their products to interact directly with operational programs in aid of communication between programmers and users and among programmers themselves.[26]

At about the same time, the Danish computer scientist Peter Naur (who as the creator of the first of the modern general-purpose programming languages, Algol 60, could be said to bear some responsibility for the "structured programming" trends of the 1980s) expressed some additional key ideas regarding the programming process itself and the relation of the programmer to the program.[27] Naur's concept is very different from Brooks's in some important respects, as it focused on democratic teams of programmers and came from a Scandinavian context of explicit concern both for workers whose work would be affected by computer implementations and for the social situation of programmers themselves.[28]

Based on his observations of actual programming and program maintenance in both academic and commercial sectors, Naur concluded that programmers responsible for the development of large programs developed "a certain kind of knowledge" through their close connection with their programs which enabled them to carry out maintenance and the ongoing changes that are inevitably required in software systems. To characterize this knowledge Naur adopted Gilbert Ryle's notion of a "theory," by which he meant "the knowledge a person must have not only to do certain things intelligently but also to explain them, to argue about them, and so forth." "Theory" in this sense is explicitly practical knowledge, depending on "a grasp of certain kinds of similarity between situations and events of the real world," which is "the reason why the knowledge held by someone who has the theory could not, in principle, be expressed in terms of rules."[29] Naur argued that such a theory must transcend what may be explicitly embodied in documentation for the program, because it (1) includes the programmer's knowledge of "how the solution relates to the affairs of the world that it helps to handle"; (2) includes the programmer's intuitive justification for every part of the program text; and (3) includes the programmer's ability to modify the program "so as to support the affairs of the world in a new manner. . . . It only makes sense to the agent who has knowledge of the world, that is the programmer."[30] Clearly, Naur is talking here about familiarity not only with the program code itself but with socially constructed knowledge. Without this kind of knowledge, Naur argued, continuing maintenance of a functioning program becomes prohibitively expensive and eventually impossible. This is vividly expressed in Naur's view of the life cycle of a program:

> A main claim of the Theory Building View of programming is that an essential part of any program, the theory of it, is something that could not conceivably be expressed, but is inextricably bound to human beings. It follows that in describing the state of the program it is important to indicate the extent to which programmers having its theory remain in charge of it. As a way in which to emphasize this circumstance one might extend the notion of program building by notions of program life, death, and revival. The building of the program is the same as the building of the theory of it by and in the team of programmers. During the program life a programmer team possessing its theory remains in active control of the program, and in particular retains control over all modifications. The death of a program happens when the programmer team possessing its theory is dissolved. A dead program may continue to be used for execution in a computer and to produce useful results. The actual state of death becomes visible when demands for modifications of the program cannot be intelligently answered. Revival of a program is the rebuilding of its theory by a new programmer team.[31]

At the heart of Naur's argument is the justification for treating programmers well in the workplace and ranking their skills high. Naur's notion of democratic programming teams is very different from Brooks's "great designers," but I have explored both points of view at such length because, in spite of their philosophical differences, both ways of thinking about the programming task reveal the recognition by computer programmers and programming theorists that the specialness of computer programming lies in the actual practice of programming and in the recognition of the program itself as practice and as an entity.

Programmers at Work: Fields of Dreams

This study makes use of observations of the programming practice of small teams of advanced undergraduates and graduate students taking an SE course in the early 1990s, situated temporally at a point when strict disciplining of the programming task was still expected for commercial contexts but beginning to be affected by some of the innovations mentioned above. The course, originally designed by Brooks, was intended to give students a taste of team programming in which they attempted to fulfill the wishes of a Client (a faculty member with a programming task that needed doing) under the supervision of a Boss (the course instructor, concerned with reinforcing lessons from the class and facilitate access to scarce resources) and with feedback provided by a Review Committee (a panel of graduate students from an advanced programming management course). Within the team, the only assigned roles were those of Producer (in essence, the manager of the team, charged with obtaining resources and keeping the team on a schedule) and Technical Director (responsible for the overall technical excellence of the result); remaining members of the team were Programmers.[32] The project had to be completed in a semester, and it had to be a project that demanded significant effort from the students. Experience of the course conferred on students a share of symbolic capital that was highly negotiable in the job market, but it was also sought out simply because it was believed to incorporate the best and most programmer-friendly aspects of the SE model:

> PROGRAMMER P: For a long time I had heard about this class being really hard. . . . I was really looking forward, though, to working with the group and getting something really big done. . . . [T]hey want you to do something that at least appears to be useful.

> PROGRAMMER E: The purpose of the course I saw was to have a project, a product that you take from start to finish. . . . My hope

> was to gradually do the design, and build up, and then work on the project, and then finally have a nice, um, steady collapse, I guess, to the end.[33]

Throughout the semester the students were observed by a team of anthropologists, who collected extremely detailed information, including audiotapes of formal and informal meetings, programming sessions, and interviews, as well as drawings, printed class submissions, program printouts, and other artifacts produced by the team in the course of the process.[34] I examined the activities of one specific team in detail for this article, although I reviewed the notes and artifacts relating to observations of other teams for context and comparative purposes. Finally, I viewed the program produced by the team in operation while I conducted an informal interview with one of the team members.

In the still-sparse literature on the empirical study of programming, complaints have been heard that most of the hard data available are based on student programmers and are inapplicable to real-world programming practices by professional programmers.[35] What little evidence there is has been gathered for the purpose of improving programmer efficiency, and until recently there has been little concern with the social aspects of the activity or the intrinsic importance of its social situation. Thus the original project constituted an important contribution to understanding real activities and attitudes of programmers confronted with fitting into specific regimes of practice. For me these rich data, reliably gathered and well documented, constituted a window into a specific moment and provided the opportunity for further investigation of programmers' engagement with their medium.

I selected a single particular team for study here because most of its members, all graduate students, had had prior professional programming experience in a commercial setting. For this reason it was possible to compare observations of the team with at least one other contemporary full-project study of professional software designers to get a feel for its similarity.[36] The specific social and professional situation of the programming team was also very important to the way they proceeded and to their relationship to the pedagogical context. The original investigators noted that this particular team essentially disregarded the context and proceeded with what seemed to be their own goal of achieving an aesthetically pleasing and technically sophisticated program.[37] This conclusion was soundly based on the testimony of the programmers themselves:

PROGRAMMER P: I knew I had to take the class, so instead of making something boring and stupid that I'd never use again, why not make something that was neat and that people would say, wow, you made something really neat and something useful, too, something that would help me in my work.

PROGRAMMER T: I wanted to have a program that would let me . . . [do] some of the stuff that I, I felt like doing. . . . So I guess, uh, I guess that, I mean, pa . . . a big part of the project was my idea to begin with, and then we went and found [the] Client to be willing to . . . to do it for us.

PROGRAMMER C: Well, we wanted to make a cool [program]. We got together before [the course] started, we were [unintelligible] team . . . so we were all pretty fired up about it.

Although it was not unheard-of for projects to be so arranged in advance, clearly this project functioned from the first much more like an in-house industrial programming project than did the others, which were picked from a list by individual students and thus were grouped by chance. The members of this team, on the other hand, were able to make use of the project for their own purposes because their activity was also strategically situated in a partly congruent but much larger field. Three of the four members of the team were or had been research assistants in the same specific laboratory setting for which the program they wrote was designed to serve as a tool, and their Client was not only a principal researcher in the laboratory but a recognized authority in the field, so the team's efforts could be understood as well as sanctioned by him. The program was an original effort, something not previously attempted by anyone in the world, and the programmers were aware of this fact.[38] The team was not paid for its work, but if they were successful, they stood to acquire the additional social capital of adding to the distinction of the laboratory. Their program was demonstrated effectively to several visiting dignitaries toward the end of the project and was subsequently made a feature of the laboratory's exhibit at a major research conference. The published proceedings paper is still cited twenty years later.[39]

The laboratory setting was also significant because of the specific tools that it provided the programmers to work with. This included both state-of-the-art hardware and a set of software tools for interfacing with the hardware that the programmers could take for granted; the program they wrote drew upon the software support of libraries of routines and procedures that had been developed in that laboratory. In fact, some of it had been worked on by some of the programmers themselves, giving them the benefit of the kind of special knowledge Naur discussed.

Thus the team saw their project not chiefly as a class project but as one that would at the least be used in an ongoing research setting and that at the most might receive favorable national notice. In one respect this can be seen as setting even more stringent requirements for their work, although the requirements would be of a specific kind: elegance, economy, and modifiability. They could assume that their peers—and, even more to the point, their superiors—might actually see the program code they wrote or at least that they would, on viewing the program, be able to "see through" the interface to understand the success or failure of their implementation even without seeing the code. The levels of skill used here were therefore certainly higher than are called for in the average real-world programming task. The programmers were in no doubt themselves about this:

> PROGRAMMER P: Our project, we four, I hate to say it, but we were highly qualified to work on it. That's one of the reasons why it was such a resounding success, was because the right people for the right job is what you need. . . . We were good. We were some of the best people, like, around, I wouldn't say in the world, but [at our institution] we were some of the best people that were qualified for that.

This team was in a peculiar position in relation to their Boss, both because they were much more skilled in the specialized area in which they worked and because of their special relationship to the work of the laboratory, to which he was a relative outsider. This played itself out in a relationship that developed much more as peer-to-peer than the relationship developed by the Boss with other teams. The following conversation took place during the last formal meeting with the Boss:

> BOSS: I think this one is real exciting.
> [Programmers brag about reactions of others and their plans to work with the program after class is over.]
> PROGRAMMER P: Yeah, I think it's going to [be demonstrated at a computer conference] and it's going to be used to impress some people there.
> BOSS: Real good. Put a footnote on the handout that says Boss . . . on it.

The laboratory connection and the programmers' awareness of working on an unusual project of potential significance and visibility were clearly fundamental to the formation of group identity and loyalty, in spite of some internal struggle that emerged toward the end of the project. As the program began to be demonstratable, both the Boss and the Client tried it out, and the Client even invited others—eminent researchers for whom the Client himself had respect—to look at it:

> PROGRAMMER T: [Visitor 1] and [visitor 2] came and they seemed pretty impressed and everything and then they left. Then [visitor 1] came back. Like you know a minute later he comes back and says, "Well you know I did something like this and this is what I had."

Group members then began to talk up their success as they worked against time to complete the project to turn in on the last day:

> PROGRAMMER P: Everybody has been telling me [unintelligible], really everyone has been telling me that they thought our project really is taking off.
> PROGRAMMER T: I've gotten quite a few comments like that too. . . .
> PROGRAMMER P: What did [visitor 3] think? He's more important than the other guy.
> PROGRAMMER E: He said it's the best . . . project he's ever seen.
> PROGRAMMER C: Or he's seen in years. That's what he told me.
> PROGRAMMER P: Oh that's fantastic.

Programming as Process: Programming Models and the Timeline

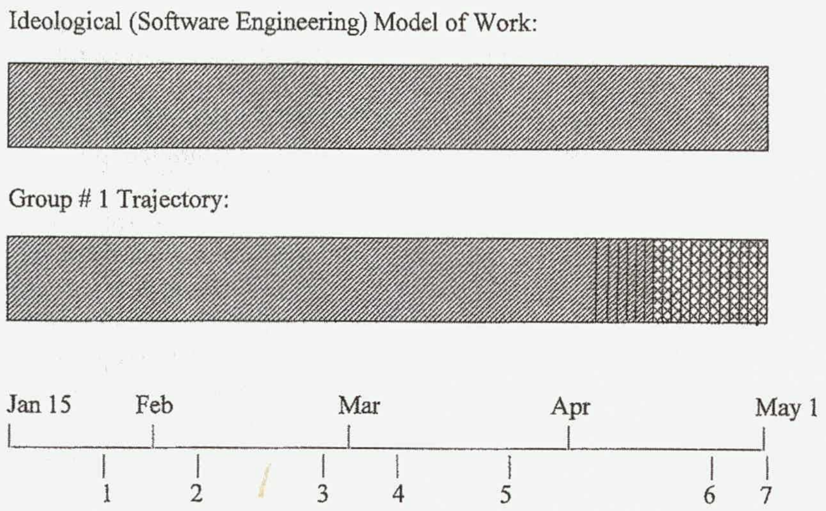
I have already mentioned the classic, orderly, sequential, "waterfall" model of the idealized software engineering process. In its various manifestations, the model generally maintains that there is an orderly sequence to the design and programming of a software system, and that sequence consists of the general steps of problem definition (requirements analysis), problem solution (program design), implementation (programming), testing (debugging), and maintenance. (Note that an informal version of this model has been internalized in the first citation above, from Programmer E, who had had prior experience working on large commercial projects characterized by a degree of division of labor.) The model is so called because each step must be completed before the next step can be undertaken. It is recognized that each phase may go through several iterations of "stepwise refinement," whose general guiding principle is to break down the problem into smaller and smaller fragments until it has been satisfactorily solved. And there is a perennial discussion among computer scientists and managers about how much time each phase of the process should take; Brooks favors one-third planning, one-sixth coding, one-quarter component test, and one-quarter system test, or fully half the time to be spent on system testing.[40] In practice, however, this ideal is almost never met; hence, software engineers are frequently advised to plan in advance for deliverable subsets of promised systems in order to keep customers happy while the inevitably delayed testing is completed.[41]

Other researchers, bearing the effects of the SE model in mind, have made several important observations about these programming teams. The researchers' studies focused upon the programming process as socially situated action, and they have been particularly helpful in identifying several aspects of the process neglected in management-oriented studies that pay attention only to how well or ill programmers are meeting their SE-dictated milestones. First, the researchers noted that the programmers were not motivated throughout by the monolithic goals of the bureaucratic SE model for solving problems and completing product logic but rather pursued individual and group goals, such as institutional rewards and struggle for personal power in the context of the programming task. The team discussed here was on that measure the most professional of the teams observed in that their concern with product logic governed more than three-quarters of the time spent.[42] Second, the researchers observed that during the course of the process itself the team members as individuals and the teams as collectivities developed their own, sometimes changing perspectives on the purpose at hand, constructing these perspectives in the undisciplined space still not yet "tamed" by the "science" of computer programming.[43]

These conclusions have been echoed from another perspective by a contemporaneous report of an observational study of a professional software design team.[44] This team, which carried out design only, not programming, was not constrained to any particular development methodology, but they were working in an OOP environment on an infrastructural project. Diane Walz and her colleagues reported that, during the course of the design work, which lasted four months, there was a shift in perspective. At the beginning, the designers worked to gather knowledge they would need to carry out the design; they then moved on to establishing in some detail the users' requirements for the system; finally, the designers shifted to a concentration on specific design approaches and the establishment of a concrete design suitable for implementation. Of particular interest, since this study does not pay much attention to the noninstitutional purposes of the designers, is its observation of the phenomenon of "shutdown": the designers simply ceased to specify requirements further, even though requirements were not completely clear, at a point about halfway through the whole project, simply going on to the next phase with what they had in hand. The researchers noted that this phenomenon has been observed before in other such studies and that it tends to happen when team members become aware that only half the allotted time remains. We will see that it appears in the project studied here as well.

The progress of the student project was regulated by an externally specified timeline marked off by required deadlines for the completion of specific documents and their review by a Review Committee. The specific documents were a Project Definition, a User Manual, and an Implementation Manual, and final versions of the latter two had to be submitted with the completed project at the end of the course. The Project Definition was due three weeks after the beginning of the course, and the User Manual was due two and a half weeks later; the Implementation Manual was due three weeks after that, but a two-week spring break intervened. About five weeks later the finished program had to be demonstrated and presented formally, and a week after that the program and all final documentation had to be turned in. Although the students were encouraged to develop additional milestones for themselves, and they made periodic short-term efforts to do so, clearly the externally imposed milestones were what really drove the activity on the project. In this it was more constrained than projects carried out by contemporary professional programmers, who in cases of schedule slippage are sometimes able simply to reschedule.

The students were similarly encouraged to begin their work by gathering everything they needed to know to do the project. It is a remarkable commonplace of the computer programming field, driven by the speed and ubiquity of change, that its participants expect to have to learn new things that they do not already know for every project—and this is not limited to domain knowledge of the application field. Although in the business world an effort is made to put together programming teams with the requisite experience and skills for particular projects, it is assumed not only that additional skills may be needed but that the programmers will be able to master them in what may seem to the outsider a very short time.[45] This mastery of a set of learning skills



Project Milestones:
1. Project team assignments (January 22)
2. Project Definition due (February 7)
3. User Manual draft due (February 26)
4. First class presentation (March 7)
5. Implementation Document draft due (March 21)
6. Final class presentation (April 24)
7. Final product delivery (May 1)

Product Logic	Institutional Demands and Rewards	Struggle
▨	▥	▩

Figure 1. Timeline and observed group activity frame deployment (derived from Holland, Reeves, and Larme, "Constitution of Intellectual Work," fig. 3).

is at least part of what full qualification as a programmer means, and it reinforces the craft quality of the work.

The student programming team was no exception, in spite of the fact that they felt themselves so well qualified. Their first formal project meeting, held about a week after the start of the course, consisted of a detailed presentation by the Producer and Technical Director of the programming environment they would be using and the existing program libraries they could draw upon for low-level routines. In addition, they attempted to settle on standards for their own practice as a team, including the use of a version-control repository that would enable the whole team to work simultaneously on various parts of the program without getting in each other's way. A few days later they visited another installation to work with a program somewhat similar to the one they planned to write, and a few days after that their Client introduced them to another expert with whom they were able to discuss design ideas. They then prepared and submitted the design document to their Review Committee; several days later they attended its review and then revised the document for formal submission in class. During this period it was clear that the team was functioning primarily in tutorial mode; although they did discuss their own ideas (and, as we have seen, they had already been discussing their own ideas for some weeks before the class began), they concentrated primarily upon gathering ideas and information from others.

The next phase of the process might be characterized as brainstorming, although certainly it was the case that nearly every meeting of the team was characterized by dialectic interchanges. Specifically here their work was focused initially upon the functions the program should incorporate, which were discussed generally from the point of view of their external appearance to the user of the program but which could quickly veer off into more detailed technical issues. A few days into this period, in a regularly scheduled meeting, they were asked to provide the Boss with an informal design document outlining the "guts of the program" the following week. Although none of the meetings that followed during that week were recorded, it was remembered by the team as "the week that we met tons," and the artifacts they produced during these meetings portrayed detailed work on both the external interface between the program and the user as well as details of program internals: logic, interfaces with libraries, data structures, and other basic technical matters of the kind that the Boss had requested.

After another week of similar activity the User Manual was submitted to the Review Committee along with a set of time logs that were required to be kept as part of the pedagogical effort to develop internal discipline. Dorothy Holland and James Reeves have shown that the programmers ignored the intent of this requirement, using the after-the-fact preparation of the logs as an exercise in team memory.[46] It is clear, however, that the episode also helped the team create a communal memory and forced them to situate themselves on the timeline, thus establishing a perspective on the work they had carried out to that point.

This became especially clear as the team worked through the following week leading up to spring break, when they began to develop an urgent desire to have some kind of skeleton program running before the break. Part of this motivation was certainly due to the deadline for the Implementation Manual, which was due a few days after the break, since most of the team members were planning to take at least some vacation time. Under this specific pressure (the Implementation Manual was to contain the technical details of the program logic and its machine implementation; as a document its purpose was to provide the information necessary for other programmers not familiar with the program to maintain or modify it), the team was moving to what I would call its first shutdown phase. That is, they were trying to make the major implementation decisions that they would put into practice thereafter, and they were trying to implement and run enough of their prototype to be sure that their design could be made functional as a program.

After the Implementation Manual was submitted, the team went into a full-time coding mode, and two weeks later they were able to demonstrate the program for the Boss. After this first demonstration, the team focused individually on several of the foundational elements of the program. After a second demonstration for the Client a week later, the team began to be concerned especially with integrating the parts of the program to work together. As the team moved toward a "code freeze" in two weeks, timed to just before the formal presentations, integration became their most important focus, and they were able to achieve an impressive presentation because they focused carefully upon the complexities of the presentation setup as well. Once the presentation was successfully completed, the team spent their last week of formal work improving the program and preparing the documents for final submission.

Programming as Technical Activity: Communicating Technically

Clearly this sequence of activities shows that members of the programmer team discussed here were constrained, as all groups working on time-limited tasks with external supervision are, by their circumstances. These circumstances forced them to carry out tasks they thought were pointless, to do things in what they viewed as less than optimal ways, and to work harder than they sometimes liked. What was especially striking here, in view of the arguments that have been put forward by Brooks, was how the "bandwidth problem" played itself out. It has been argued that one of the major sources of overhead in group activities in general is the attainment of sufficient quantities of what John B. Smith refers to as "shared intangible knowledge" for the group to function effectively.[47] Computer programming carried out as a decomposed task, with the assignment for different members of a team to make different parts that must interact, is an especially emblematic example, and the difficulties of this communication task lie at the heart of Brooks's "mythical man-month."

Programmers agree that in an ideal world all programs would be written by a single person, precisely because it is so difficult to establish the required level of communication, but usually time constraints or sheer complexity require that a team undertake the task. Although commentators on this problem frequently write as though they thought programmers have somehow to communicate to each other everything they have learned since birth and have no external physical and social world they can mutually reference, the task of arriving at even the smaller body of knowledge that constitutes a problem domain is not trivial, and it accounts for the "high-bandwidth" methods of communication frequently used not just by programmers but by many knowledge workers: drawings, enactments (e.g., Holland's "air demos"), and imaginary narrative scenarios. The whole beginning, tutorial part of the programming team's task, in fact, can be looked upon from the engineering point of view as establishing a store of "shared intangible knowledge." Team members made copious use of whiteboards for drawing diagrams of program structures as they agreed on all kinds of basic conventions, including even how they liked to use white space in the layout of programs on a page. They carried out information-gathering tasks, and they talked interminably about various programming conventions to establish shared understandings.[48]

Yet they did not even try to share everything, because they didn't have to. They depended upon one another's expertise, which was established in the first weeks through their exploratory discussions, and they especially depended upon their mutual acquaintance with and preferences for not only programming techniques but also a world of existing programs that served them much as literary intertexts. This was particularly evident when individual programmers' allegiances led to a clash of cultures. The following example comes from an interview, but the attitude was played out throughout the course of the project as the programmer in question continually positioned himself as the user advocate:

> PROGRAMMER C: I think that most computer scientists really don't care about user interfaces at all, I mean, they want code that

> works, that does something, and user interfaces are like, you slap it on at the end, especially in environments like UNIX where you can make any text interface that you want that's as hairy and moronic and non-intuitive, and as long as it gets the job done UNIX heads will love it because it works, and it's just as arcane as every other UNIX program, whereas I, someone who's in love with the Macintosh interface and the Macintosh philosophy . . . I think the user interface is a lot more important than most CS people think.[49]

Other programmers, throughout the course of the project, focused positively and negatively on specific features of programs they had seen or used when talking about features they wanted to add to their own programs.

Programming is unique among engineering disciplines in that programmers have at their disposal an especially powerful means of externalizing their thinking: they write program code to embody it.[50] As Holland and Reeves have observed, "While [software] engineering talks in terms of inanimate objects, programs behave like agents."[51] Thus, if their authors have not succeeded in interacting effectively, the parts of a decomposed program will not do so either, and the program will not run or will crash. This absolute desideratum is so important to the programmers themselves that exponents of the SE model inevitably fail when they try to persuade programmers to design completely before they code.[52] The team observed here began talking about actually writing code as their tutorial period ended, before they had even finished revising their Project Definition, and were supported in this decision by the Boss:

> PROGRAMMER P: I think that we've found at least two solutions that are worthy of investigation. And I think we need to do a little bit of experimental code before we . . .
> BOSS: Okay, watch out that in dealing with that you don't wind up with a nasty NP-complete problem of some sort.[53]

In fact, anyone listening to the talk of programmers involved in programming will have little doubt of the at least figurative agency of the programs, since programmers habitually talk about them as though they were alive. Sherry Turkle has spoken in psychological terms of a computer as a "projective medium" or a "transitional object" for a programmer, but she overlooks the fact that in a very real sense the program is active as the programmer's ally in the programming task.[54] Because the program does behave, and its behavior has real effects in the world, it is not just a way of speaking to refer to the program in behavioral terms. From the beginning, long before having written one line of code, the programming team used vivid images to refer to their program's behavior and indeed to begin to outline the behavior they expected it to have at various levels of detail:

> PROGRAMMER P: The main part of our program, um, we should think of it as, as waking up in an environment like that. Our program wakes up in a [software] environment that has already started and . . . then as soon as our important code gets called, you know, we [the programmers personified in their program] toss up our [tools] and other things and let you [the user] start working.
> PROGRAMMER T: You know, they [the user] just call some specialized routine that we wrote that would just look at that, that information we saved and say, "Oh, they want to get to arm rotation." And so then it would look up arm rotation in the table and say, "Oh, it's in structure body number, you know, number 5." And so it would step through and return that corner so that they could actually use it.

These ways of speaking are metaphorical and subject to error until they begin to refer specifically to observable, running programs, and programmers' awareness of this problem is reflected in the instinct to "go to code" as quickly as possible, a practice appreciated by Brooks. This example of an exchange between one programmer carrying out ordinary tasks and the Producer trying to pursue the orderly SE model was taken from just after completion of the Project Definition. The Boss has heard it all before:

> PROGRAMMER E: I really don't see why we're talking about this now. . . .
> PROGRAMMER P: . . . [I]f somebody just implements whatever they feel like, everybody else is going to spazz if it doesn't react correctly. I mean, I'm willing to, I'm definitely willing to make up [unintelligible] as long as we say do it now. . . .
> PROGRAMMER E: Then why are we even having a meeting at this point? We can just implement the stuff we've already decided upon, and then, you know, it could be this problem will clear itself up after we [unintelligible].
> PROGRAMMER P: Right now I'm very skeptical of that. That's what this week is for. Deciding what we're going to, what we're going to implement before we do it.

> PROGRAMMER E: Okay, so, yeah, let's just say we decide all this stuff and then the first day we try to actually write some code, we find out that everything we decided is completely void and we have to go down [unintelligible] trash, and then we're screwed.
> [Shortly afterwards the Boss arrived for a scheduled meeting.]
> BOSS: Sounds like business as usual. [laughter]
> PROGRAMMER P: Well, what we've been doing for the last week is trying to work some more on planning the details. . . . And coming up with the exact details has revealed the fact that we'd all built up different pictures in our mind of exactly how it's going to work. [laughter]
> BOSS: Not surprising.

Reflecting on this period of the work after the fact, the Producer clearly expressed the attractions of programming as against planning:

> PROGRAMMER P: This was such a big project that there was tons of stuff to figure out and we eventually got sick of trying to figure everything out before we even went near the computer. We wanted to get a little bit of stuff up and running and get a feel for what would work well, and that was a good idea, but then once we got it up and running it was starting to work and we got a feel for how it worked, we just kept programming. . . . I think the other guys basically just wanted to jump on the computer and type a whole bunch and, you know, let's just get it going and make as much progress as we can really fast.

The team faced several problems due to communication failure, some minor and at least one major, during the course of their work. Nine weeks into the project, after the Project Definition and User Manual documents had been prepared for the Boss but before any part of the program had successfully run, nearly a whole project meeting was devoted to mutual clarification of the programmers' understandings of each other's terms. This effort, probably prompted by some work being done on the Implementation Manual, resulted from two programmers' having discovered, in talking informally about the pieces of program for which they were responsible, that they were using two different terms for the same concept:

> PROGRAMMER P: So, that's a group for me. What I've been calling it is an object, and so when E and I talked it's been a little difficult. . . . So now I will call it a group. . . .

PROGRAMMER E: Mainly, we all knew what we were talking about but we used different terms for them, especially in the manuals.
PROGRAMMER C: Uh-huh. Yeah. Well, I just got finished doing a successful compile of the thing and after this is over I can maybe go show you what, see if it works.

A similar example arose dynamically, during a meeting, as late as the code freeze phase at the end of the project. Here the internal complexity of the parts' interaction obscured the action of each part so that the problem could not be discovered from observation of the program alone. As one programmer talked his way through the operation of his part of the program, another realized that a misunderstanding had developed about a data hierarchy:

PROGRAMMER T: You see what I'm saying? For those times when you don't want to delete everything, when you want to leave them.
PROGRAMMER E: Oh, you're talking about deleting the child. I've been talking about the parent.
PROGRAMMER P: Well, let's draw a picture.
PROGRAMMER T: I'm talking about the . . .
PROGRAMMER C: I thought he was talking about the parent, too.

The building of shared knowledge lasts throughout the duration of a programming project, and it consists of understandings constructed in the social world of the programmers and their program, creating Naur's theory as practice. That this "feel for the program" cannot just be reconstructed from documentation was well expressed by the programmer serving as Producer when, during the formal review of the Implementation Manual, a committee member made him aware of some failings of the manual:

PROGRAMMER P: I guess someone would, like for the next . . . class if they were going to use this, I would guess they would read the user manual, go play with it for a while, and so then they understand the concepts of hierarchy and they understand that there is a [program object] there. And then they would read this and find out how we implemented the [object]. . . . Now unfortunately it's almost mandatory that a person reading this know something about hierarchical, graphical databases. . . . So come to think of it, if this is the first time the person has seen something like this . . . then I don't think that this will cut it.

During the creation of a program, details can become very fluid on a day-to-day basis as the program takes shape, such that programmers begin to depend more upon the program's behavior than upon explicit knowledge of the code that causes it. A good example comes from one event during the last observed meeting when one programmer asked another about a minor feature in a part of the program he had written a month earlier:

> PROGRAMMER C: I have one question and that is about the X and Y events. If your X and Y just are exactly the same except X is without the mouse button pressed, right?
> PROGRAMMER T: I think so but I haven't looked at the code lately.
> PROGRAMMER C: I think that's the way it works. You don't want to X event if the mouse button's pressed.
> PROGRAMMER T: I'll have to look.

The precise details of the program as code have given way to the program as agent.

It is difficult to observe the process by which this happens, because most of the dialogue between programmer and program takes place interactively, at a computer terminal, and if the programmer is using a sophisticated, integrated programming environment with built-in debugging aids, the program will no sooner offer evidence that something is wrong than the programmer will change it, sometimes in so rapid a flow of interaction that the program seems literally to be evolving smoothly. Therefore, the best examples of this sort of thing are the instances in which programmers find themselves observing a behavior that they didn't expect to see, when the program-agent demonstrates its independence.

Holland and Reeves have cited one instance observed during a programming session when a programmer working with two others, on separate workstations but in the same room, started to accuse the others of having caused a strange effect that he didn't think he had programmed, and the three together discussed the possibilities until they came up with a reasonable one.[55] A similar situation arose during the rapid early evolution of the running program:

> PROGRAMMER C: You can tell if you're going to select an old one 'cause it's highlighted.

PROGRAMMER P: Well it's gonna . . . I, I got that highlighting word yesterday, but then today I did something, highlight's gone. But I know the routine works, 'cause yesterday it was working pretty fast.

Unpredictable effects were still happening as the programmers were completing the integration of the parts of the program, preparing for the code freeze prior to the formal demonstration:

PROGRAMMER E: Remember you were commenting on the fact that when you were creating triangles and you picked the color your last selected vertices went away?
PROGRAMMER C: Yeah, how does that happen?
PROGRAMMER E: I was thinking about . . .
PROGRAMMER T: Oh, that's because . . .
PROGRAMMER C: As soon as you click a color it sends an off to the triangle. Do you think it would screw anything up if [unintelligible] the long command did not send an off event?

Here the programmer who actually had the original question now has enough information about the other parts of the program to request a remedy from the programmer whose program's behavior is causing the effect.

Programming as Social Activity: Romancing the Program

This last example brings up perhaps the most interesting aspect of computer programming, the one most resistant to "rationalizability": the fact that the programmer, in ways that Naur's concept of theory as practice suggests, retains a kind of ownership of or bond with the program he writes that is very difficult for another to reconstruct or recreate because it is part of the process of what Brooks calls "growing" programs in the highly interactive settings of programming teams using high-tech development tools.

The theme of ownership began to emerge in the programmer team meetings from the time when actual programming started during or just after spring break. The programmers began to speak on behalf of the portion of the program for which they were responsible, as in this example, when one of them refused to make decisions regarding other segments of the program:

PROGRAMMER P: That's actually outside of my domain. . . . You guys can do what you want.

Later in a review meeting he refers questions from the reviewers to another programmer, counting on their understanding of this as not just a case of passing the buck:

> PROGRAMMER P: Good question. The member of our group who wrote a lot of this stuff we've been talking about isn't here. . . . T is the one who did the [program part].

Toward the end of the project, in the intense period of work that preceded the code freeze, it was necessary to redistribute some of the tasks so that the project would be completed on time. Since by this time the personal attachment the programmers had invested in their code matched the deep knowledge they had of the parts they had been working on, the Producer and Technical Director discussed the reapportionment of tasks with evident considerateness:

> PROGRAMMER P: Well, the thing is, I'm worried about piling too much on E too, but the thing is he's the one who's thought the most about how those commands would work.
> PROGRAMMER T: Well, should we give X and Y to someone else?
> PROGRAMMER P: Okay, how about then C gets X and Y. I think that would work out okay. He's been working with coordinate systems and stuff. . . .
> PROGRAMMER T: That's probably good, that's probably . . . [addressing P] Do you think so? I'm saying that's probably good and that's like your work, right?
> PROGRAMMER P: Right, I don't mind doing that.

Similar consideration was being shown as the demonstration drew closer:

> PROGRAMMER P: You two [E and C] need to make sure that you both understand how the X/Y is being split into two, and you [E] probably need to impart some knowledge to C as to how you like to Y stuff.
> PROGRAMMER C: He's imparted that, such knowledge, and I'm currently trying to implement it.
> PROGRAMMER P: Wonderful, wonderful. Wunderbar. That's good.

Finally, in the last week the Producer advised the programmers to help one another using a self-deprecatory example:

> PROGRAMMER P: If I was stuck on some bug, especially if it had to do with some other part of the program, I'd go get the person

> who was involved. . . . So don't hesitate to work in pairs. Sometimes with the transformation stuff I really screwed up your stuff. Boy I messed up scales for a week once.

As is the case in any team effort where tasks must be parceled out, programmers' personal identification with their individual assignments can become a liability to the team effort as a whole if it leads to manipulation of team resources for individual purposes. This played itself out overtly here in the form of an apparent shift in power toward the end of the project, particularly during its last month, when the team was fully absorbed with integrating the parts of the program and preparing it for presentation. Through the first two and a half months the meetings were clearly directed by the Producer, whose efforts to keep the team on track as far as the process was concerned were reasonably successful and whose enthusiasm and good humor kept the meetings moving, even when they did not go as he wished. But once the coding task became serious, the Technical Director, who had not taken the lead up to that point and had been nearly silent in Boss meetings, moved into overt control of the meetings and began to attempt to exert the authority to make final decisions that the SE model would have granted him. Although a careful reading of the transcripts shows that the other members of the team tended to disagree with the Technical Director's decisions and even apparently to ignore them, no serious degree of disagreement emerged in the meetings.

The final interviews with team participants, however, told another story. The Technical Director was seen by the other team members as pursuing his own goals, and they all resented it. But the way this was expressed—and presumably experienced—was in terms of a sort of proxy battle of the participants' program parts. The Producer and Programmer E were particularly condemnatory when the Technical Director made changes to some of the program segments initially assigned to the Producer, segments on which all the others depended and changes that they saw as benefiting the Technical Director's program segment at the expense of the other members' work. Interestingly, although elsewhere in the interview the Producer portrayed the group as united in opposition to the Technical Director during the period of his dominance, here he was rather reticent, perhaps because he also saw himself as at fault:

> PROGRAMMER P: So, T at one time was impatient and took some of the database functions over that I was going to implement, and

> he implemented them, or he even altered some of the ones that I had to make them work better with his code. Then they didn't work well with anyone else's. So, it was when people pretty much stepped out of their bounds. . . . When T implemented X and Y, which were fairly complicated things, I worked on those some and problems resulted from it because he understood them better than me.

The problem dominated Programmer E's interview, as he returned to it several times with more and more emphasis on ownership with each repetition:

> PROGRAMMER E: Somebody had written a lot of code, and then another person went in it, and then changed it. . . . Made massive changes. And, um, broke [some of the] things . . . the other person had done. And, um, he hadn't asked the other person if he could go in. . . . [A]ll of a sudden somebody comes in and changes it out from under his feet. And then he comes in to work, continue working on his code, and his code isn't there anymore. There's somebody else's code.

Programmer E portrayed the conflict as a struggle between the Producer and Technical Director; interestingly, he blamed the Technical Director for carrying out what was precisely his assigned task: exercising authority over technical matters. Programmer E portrayed himself and Programmer C almost as standing on the sidelines watching the struggle. Programmer C, on the other hand, made almost nothing of the struggle, remembering only the user interface issues that interested him in particular:

> PROGRAMMER C: We had a few conflicts just over user interface issues because, like T thought that this was inconsistent with this other user interface thing, or he thought this should be this way, and that other tool should be modified to fit his favorite method, and that was sort of a drag, but I think we all, he compromised most of the things.

There were clearly many aspects of this struggle that had to do with personal relationships, which in a very real sense were expressed through the eventual shape that the integrated program took. The particularities are not so important here as the return to the notion that the program or program segment is a (perhaps relatively innocent) agent in a social context, and its behavior is used by programmers as an index to the intentions and preferences of the programmer who wrote it. Clearly there is a tacit etiquette to the ownership of program code in team projects, which programmers transgress at their social peril: you may not alter the code of a colleague without his or her permission, just as it is not acceptable to discipline another person's child.[56]

The issue of ownership brings us back to the question of the programmers' being separated from the product of their labor. The course the programmers were taking required them to sign over ownership of their program to the Client at the outset, but it is especially interesting that the Boss felt called upon to discuss this issue with the programmers in their last formal meeting with him, as they delightedly envisioned themselves doing further work on the program after the course was over:

> BOSS: One of the points of having good implementation documentation for follow-on is that this project has a life of its own after the end of the course, whether you're involved in it or somebody else is involved in it. . . . I mean I think this one will have a life of its own.
> PROGRAMMER P: Yeah, this one definitely will.
> [The programmers mention numerous features they dream of adding, and the Boss urges them simply to add a "wish list" section to the Implementation Manual.]
> BOSS: . . . Because these things, especially something like this, is never finished and needs a . . . It's a vehicle for experimentation and future investigation, so it needs to be able to continue to grow in that way. . . .
> [Programmer P indicates that he hopes to do official work on it in the laboratory.]
> BOSS: . . . Of course there's nothing wrong with the Client contracting with the people who developed it . . . but when it ends as a [class] project its future becomes up to the Client.

The discussion here further emphasizes the extent to which the programmers on this project enjoyed an unusual relationship to their program based upon their connection with the laboratory. As the matter developed, all four of the programmers did additional work on the program to prepare it for demonstration at the computer conference a month after the course was over, three of the four did some further work on it, and one continued to work on it for several more years.

In the follow-up interviews, the programmers were specifically asked about their thoughts with reference to the ownership issue. The Producer's take on this was an interesting example of mixed feelings, in which the partially assimilated SE model was paired with a more plangent theme of separation:

> PROGRAMMER P: I've learned that it's nice to be able to say, yeah, it's mine, but then you have to support it. . . . [I]t's good not to be tied to it. Because I still get the recognition. . . . The equipment we work with . . . we're talking about tens of thousands of dollars, up to like a million dollars. . . . You kind of donate the idea to the people who gave you, who loaned you the equipment, but still it's frustrating, as a programmer to come up with some new innovative thing. . . . You created it, you would like to be able to control what gets done to it, what gets added to it or removed from it, who it's given to and sold to, how it's integrated into some other product. . . . I may see somebody really, in my opinion messing up the code, now it's so disorganized and I'm so mad, all our hard work has gone to waste. . . . I got a grade, I got some credits, and they got a program. So, it's like a deal.

In the response of the Technical Director can be seen the theme of the loss of the program's theory as it continued to evolve without him:

> PROGRAMMER T: At the beginning of the summer it was still, you know, I knew how things worked well. . . . I know . . . again how things work pretty well, but, uh, when I first started working on the stuff for this project [to adapt the program for a new machine] a lot of stuff had changed while I was . . . while I had been ignoring it. . . . Um, C had added a bunch of things . . . and it wasn't quite the program I left.

Programmer C was still working on the program when he was interviewed and was in the process of understanding the parts of the program that he did not write:

> PROGRAMMER C: But it's really sort of my baby now, I think. I mean, most of the code is not mine and it's not like I can claim to even understand it, much less have written it, I mean, there are a lot of things where, I mean I was really worried about finding some of these bugs that cropped up after it was ported to [the new machine]. [S]ome things just broke and I was like, it's going to be

impossible for me to find those bugs, I don't even know how this stuff works, but it turned out to be not that hard.

Yet it was "not that hard" precisely because he had been part of the process that built the program's theory along with its code. And it is clear that the pride of ownership in these mind children lingers. In preparation for the work reported in this article I was given a demonstration of the program by one of the programmers, not then working on it, who dutifully showed me all its major features. When in trying it out I accidentally failed to make more than perfunctory use of what I later realized was the central feature he had programmed, he made a point of demonstrating all the program's capabilities.

Postscript

Naur has argued that there is a need to recognize this social and practical connection between the programmer and the program and to incorporate it into the treatment of programmers at work: "On the Theory Building View the primary result of the programming activity is the theory held by the programmers. Since this theory by its very nature is part of the mental possession of each programmer, it follows that the notion of the programmer as an easily replaceable component in the program production activity has to be abandoned. Instead the programmer must be regarded as a responsible developer and manager of the activity in which the computer is a part."[57] Although Kraft's observations regarding the deskilling of programmers have been contradicted both by the fortunes of SE in general and by the observations here that have demonstrated the irreducible connections between programmers and their work, it is likely that significant reform in the understanding of the programming activity will be necessary before computer programmers can be removed from the demonized category of a management tool or the demon-ridden category of a wielder of SE silver bullets. The movement toward participatory design of software in Europe with which Naur was connected has aimed at an explicitly liberatory application of computer program design and the talents of programmers to the improvement of working conditions in industry.[58] An indication of its emerging influence in the United States around the time of the observed programming project can be seen in the appearance of a paper on this topic in the October 1993 *Communications of the Association for Computing Machinery* devoted to orchestrating project organization and management.[59]

Only a little later an even more radical answer to this evergreen crisis in programming emerged. In the spirit of participatory design and built on top of many of the programming principles that emerged counter to strict software engineering disciplines, an approach to programming project practice billing itself as new emerged as "lightweight" or "agile" programming around the late 1990s and especially after the turn of the century. Nurtured in small companies where fast-evolving software products were created and at a time of economic precariousness, its purpose was to cut through heavily structured program production processes, to recognize that the production environment was innately uncertain, and to restore the enjoyment of and pride in programming practice.

The various versions of this method all adhere to the few points of the Agile Manifesto, which states that the adherents value individuals and interactions over processes and tools (including features like team self-organization, motivation, collocation of team members, and pair programming); working software over comprehensive documentation (software demonstrations and code are considered to be better representations of the software product than documentation); customer collaboration over contract negotiation (continuous involvement of customers in the full process of development); and responding to change over following a plan (recognizing the volatility of computer environments and business plans calls for a continuous development mindset).[60]

The two leading representatives of this method, Scrum, directed toward project management, and Extreme Programming, directed toward programming practice, agree on most practices and emphasize that methods used must be simple—like extreme sports, they ironically attempt to strip away layers of technology. Depending significantly on programming tools and a versioning repository for the code that is created, they instantiate the rest of the practice in physical form: customer "stories" (use cases) and the tasks generated by the team to respond to them are written by hand on cards and posted on a wall; regular meetings make use of a whiteboard for planning; programming teams work in a single room on tables pushed together; actual programming takes place carried out by pairs of programmers at shared machines. Large programming tasks are broken into segments based on user stories and tasks developed by the programming team from the stories, to be tackled iteratively until completed in what Scrum calls "sprints" of at most a few weeks. Team members meet daily before beginning work in the same room and working in pairs; tests for the functioning of code are written before or along with code segments, which are always tested before being committed to the code repository; the practice calls for the automation of testing such that as the code grows, incorporating fragments from separate pairs of programmers, the whole of the codebase is tested with each new addition so that when the end is reached there are no surprises.[61]

There is much more to agile programming than this brief outline can offer, but it is enough to suggest that the approach has much in common with all working programmers' experience of "cutting code," as we have seen in Brooks's and Naur's respective ideas as well as in the actual observation of one capable programming team, stretching thus from the 1960s to the late 1990s. The changes that took place over this time period were dependent on hardware and software development; the assignment to computers of both the deep infrastructure of power (defense, banking) and the pervasive infrastructure of information appliances (shopping, entertainment); and a process of professionalization for programmers themselves. But it seems that, as Brooks suggested and as agile programming reaffirms, the central arcana of programming have always stemmed from the same problem: that of making activity from uncertainty. As Naur argued and agile programming also affirms, this process must be embedded in social, material, and technological settings that both help constitute the process and are changed by it. The playpens of the early engineering of software have become the integrated programming environments of agile teams, but the mind children are still born in social practice and constitute expressive and meaningful behavior that contributes to the programming process. It remains to be seen whether group ownership of code as advocated by agile programming and ingrained in its practices will prove to be a chimera.

Notes

This study makes use of data gathered as part of a larger study funded by National Science Foundation Grant No. IRI-9015443 (John B. Smith, Principal Investigator; F. Donelson Smith, Co–Principal Investigator). Unattributed quotations presented in the article are taken from transcripts of the activities of a student programming team working on a major project and their faculty mentors. I would like to thank Dorothy Holland, John B. Smith, and Dana Kay Smith for providing access to these data; James R. Reeves, Anne Larme, Carole Cain, and several anonymous transcribers for their work on the project; and the programmers themselves for their good-humored cooperation in being observed. The title of the article refers to the literal embodiment of thought as offspring in Hans Moravec's robotic vision in *Mind Children: The Future of Robot and Human Intelligence* (Cambridge, MA: Harvard University Press, 1988) and to the "playpens" (private files) organized by W. R. Crowley for programmers to use for their own work within the publicly shared program libraries on the IBM 360 system programming project: see Frederick P. Brooks Jr., *The Mythical Man-Month: Essays on Software Engineering* (New York: Addison-Wesley, 1975), 133; and Brooks, e-mail message to author, December 14, 1993.

1. Lawrence Lessig, *Code and Other Laws of Cyberspace* (New York: Random House, 1999).

2. Nathan Ensmenger, *The Computer Boys Take Over: Computers, Programmers, and the Politics of Technical Expertise* (Cambridge, MA: MIT Press, 2011).

3. Dorothy Holland, James R. Reeves, and Anne Larme, "The Constitution of Intellectual Work by Programming Teams," in *TextLab/Collaboratory Report* TR92-013 (Chapel Hill: University of North Carolina Department of Computer Science, 1992).

4. Philip Kraft, *Programmers and Managers: The Routinization of Computer Programming in the United States* (New York: Springer, 1977), 22. Kraft's monograph, although impressive in a field that he accurately evaluated as being totally dominated by management discourse, is marred by a significant lack of understanding of both computer programming as a process and the tools it uses. He claims, for example, that applications programmers using high-level languages like PL/1 "need only rudimentary programming skills compared to those of the small number of software specialists who designed the languages they use" (28). In fact, the so-called third-generation programming languages, of which PL/1 was one of the early ones, require no more "rudimentary" programming skills than the low-level assembly languages and machine languages they replaced; they permit more powerfully direct expression of logical complexity because they are more abstract. Not only was Kraft's example, PL/1, one of the most complex programming languages ever devised, but one of the applications for which it was used was systems programming, certainly the most complex of programming tasks. The complexity of a problem cannot be determined in any simple way by the apparent simplicity of the tools used to solve it.

5. Harry Braverman, *Labor and Monopoly Capital* (New York: Monthly Review Press, 1974); David F. Noble, *America by Design: Science, Technology and the Rise of Corporate Capitalism* (New York: Knopf, 1977).

6. "Programmers thus persist in being something of an anomaly. . . . [T]hey are employees, but they are in a position to control much of how they will go about doing their programs . . . and to some extent even the form the final product will take" (Kraft, *Programmers and Managers*, 62). Compare these statements from a recent management-centered software engineering text: "Under time pressure to get a working system, you can use any implementation decision that meets the specification. This is one approach to prototyping: design module interfaces carefully, then use any implementation with the correct functionality, ignoring performance and capacity requirements," and further: "The problem [with defining requirements in advance] is that computers are such flexible tools that we often try to get them to do things no one has tried to do before. This often means we do not know in advance what the software should do" (David Alex Lamb, *Software Engineering: Planning for Change* [New York: Prentice-Hall, 1988], 101, 211).

7. Philip Kraft, "The Industrialization of Computer Programming: From Programming to 'Software Production,'" in *Case Studies on the Labor Process*, ed. Andrew Zimbalist (New York: Monthly Review Press, 1979), 1–17.

8. Nathan Ensmenger and William Aspray, "Software as Labor Process," in *Mapping the History of Computing: Software Issues,* ed. U. Hashagen, R. Keil-Slawik, and A. Norberg (New York: Springer-Verlag, 2002), 139–66.

9. Bruce Berman, "The Computer Metaphor: Bureaucratizing the Mind," *Science as Culture* 7 (1989): 7–42. Artificial intelligence is another field that produced a great deal of popular fanfare, at least partly due to the outrageous claims made by the earliest practitioners, claims that have not been followed up by much in the way of accomplishment. Its discourse, that of the computer model for the mind, has had much greater success; see Sherry Turkle, *The Second Self: Computers and the Human Spirit* (New York: Simon and Schuster, 1984); and, more recently, Paul M. Churchland, *The Engine of Reason, the Seat of the Soul* (Cambridge, MA: MIT Press, 1995).

10. As late as 2002, when the agile programming bandwagon had begun to overtake other models in the context of OOP practice and commoditized programming products, Kraft was still warning against management control over development practice while admitting that programmer resistance was still a force to be reckoned with. See Jacob Nørbjerg and Philip Kraft, "Software Practice Is Social Practice," in *Social Theory—Software Practice,* ed. Yvonne Dittrich, Christiane Floyd, and Ralf Klischewski (Cambridge, MA: MIT Press, 2002), 205–22. Further, the routinization theory still argued by Nørbjerg and Kraft, especially for offshored programming shops, has been rejected in contemporary studies. See, for example, P. Vigneswarna Ilavarasan and Arun Kumar Sharma, "Is Software Work Routinized? Some Empirical Observations from Indian Software Industry," *Journal of Systems and Software* 66 (2003): 1–6.

11. Bureau of Labor Statistics, *Occupational Outlook Handbook, 2010–2011 Edition,* http://www.bls.gov/oco/ocos303.htm; the latest information on wages can also be accessed from this location.

12. See Berman, "Computer Metaphor"; this critique, still repeating Kraft's arguments, completely ignored the replacement of the central control aspect of the mainframe computer model and the reskilling of computer users through the supply of autonomous desktop computers to replace terminals as all computing became increasingly decentralized.

13. Bruce I. Blum, *Software Engineering: A Holistic View* (New York: Oxford University Press, 1992), 28–29.

14. In this quotation from the observed SE course, the professor advised students to consult the paper by David Parnas and Paul Clements, "A Rational Design Process: How and Why to Fake It," *IEEE Transactions on Software Engineering, SE* 12, no. 2 (1986): 251–57.

15. Dorothy Holland and James R. Reeves, "Creativity and Rationalizability: Beasts in the Tar Pits of Software Engineering," paper presented at the annual meeting of the Society for Science and Literature, Atlanta, 1992 (photocopy in possession of the author).

16. For academic hacker culture, see James Wallace and Jim Erickson, *Hard Drive: Bill Gates and the Making of the Microsoft Empire* (New York: Wiley, 1992); Turkle, *Second Self,* 196–238; Steven Levy, *Hackers* (New York: Anchor, 1984). Examples abound of the kinds of symbolic rewards that are offered to computer industry programmers: inside the plastic case of the original Macintosh produced by Apple Computer, molded into the plastic itself, were the names of all the engineers and programmers who had worked on the project; and if the user

of Microsoft's Windows 3.1 software knew just what keys to press (the sequence was formally undocumented but was made public through numerous articles in the computer press), a "gang screen" appeared, listing the programmers who worked on the software and even jokes about the process, recognizable to their peers if not to the public.

17. Wallace and Erickson, *Hard Drive,* 385. The manager in question, Bob Metcalfe, was himself a primordial hacker, so he is not inclined to condemn programmers because he comes from the managerial side of the fence.

18. I refer to Brooks's work here in detail for three reasons: it reflects what has proved to be seen within the field of computer science as the most sane view of software engineering, based strongly in practice; it continues to be cited in reading lists put together by exponents of new programming models; and the observational data used in this study came from a software engineering class originally designed by Brooks and grounded on his evolving conception of the field.

19. Brooks, *The Mythical Man-Month,* 9–10.

20. Part of the reason for this is that many expert programmers spend time with fantasy computer gaming set in a "swords and sorcery" world, but it has entered into the common parlance of the programmer's culture totally apart from this usage. A well-known columnist writing on obscure programming tricks in the magazine *UNIX World* entitles her column Wizard's Grab-bag, and a well-known account of the emergence of the Internet by Katie Hafner and Matthew Lyon is entitled *Where Wizards Stay Up Late: The Origins of the Internet* (New York: Simon and Schuster, 1996).

21. Frederick P. Brooks Jr., "No Silver Bullet: Essence and Accidents of Software Engineering," *IEEE Computer* 20, no. 4 (1987): 10–19, quote on 10; I quote from this reprint.

22. Frederick P. Brooks Jr., "No Silver Bullet: Essence and Accident in Software Engineering," in *Proceedings of the IFIP Tenth World Computing Conference* (1986), 1069–76. It is not obvious who the editors of *Computer* meant the young woman to symbolize, for the only victim Brooks's article mentions explicitly is the "non-technical manager," whose vision of "missed schedules, blown budgets, and flawed products" leads to "desperate cries for a silver bullet." Brooks limns these managers as having pursued this quest in vain through the promises of SE.

23. Brooks, "No Silver Bullet" (1987), 11–12.

24. Ibid., 16–18.

25. Howard Rheingold, *Virtual Reality* (New York: Summit Books, 1991), 39; Frederick P. Brooks Jr., "The Computer Scientist as Toolsmith II," *Communications of the Association for Computing Machinery* 39, no. 3 (1996): 61–68.

26. This was also an important theme in *Mythical Man-Month* and one of Brooks's main reasons for arguing for small programming teams to cut down on the problem of communication (17–18). This "communication bandwidth" problem has been a major concern of cognitive scientists like Newell and Simon studying human problem solving. See John B. Smith, "Collective Intelligence in Computer-Based Collaboration," in *TextLab/Collaboratory Report* TR92-012 (Chapel Hill: University of North Carolina Department of Computer Science, 1992), chap. 4. It has been internalized as a major theme of programmer interaction at the Microsoft Corporation by its founder, Bill Gates. See Wallace and Erickson, *Hard Drive,* 303.

27. Peter Naur, "Programming as Theory Building," in *Computing: A Human Activity* (1985; Reading, MA: Addison-Wesley, 1992), 37–49.

28. Andrew Pickering, *The Mangle of Practice: Time, Agency, and Science* (Chicago: University of Chicago Press, 1995), 162n6, has pointed to similar developments in the 1950s in Britain in work on "socio-technical systems" and "autonomous groups."

29. Naur, "Programming as Theory Building," 40.

30. Ibid., 41.

31. Ibid., 44. "All repairs tend to destroy the structure, to increase the entropy and disorder of the system. . . . Sooner or later the fixing ceases to gain any ground. Each forward step is matched by a backward one. Although in principle usable forever, the system has worn out as a base for progress. . . . A brand-new, from-the-ground-up redesign is necessary" (Brooks, *Mythical Man-Month*, 122–23). Note that Brooks assumes that the maintenance staff will be different from and probably junior to the design staff.

32. This business terminology for the roles was used throughout the course and provided parameters, in the students' case, by which their grade was established. Adherence to the roles was theoretically reinforced by the fact that the team grade was dependent upon the members' fulfilling their roles, while individuals were called upon to grade each other for the quality of their participation. The importance of these reinforcements to the outcome was analyzed by Holland, Reeves, and Larme, "Constitution of Intellectual Work," and was not deemed determinative in every case. The course is still taught.

33. References to the four programmers in the team are not initials; the Producer is referred to as Programmer P, the Technical Director as Programmer T, and the remaining two programmers as Programmer C and Programmer E. In the quoted material an effort will be made to remove identifying information, including clues to the details of the project, so sometimes it will be necessary to substitute or replace specific identifiable references to programming constructs.

34. The purpose of the study was to understand situated cognition in group practice to support theory building for computer-supported cooperative work; see Holland, Reeves, and Larme, "Constitution of Intellectual Work." Interestingly, the anthropologists observed that much of the informal documentation created during meetings was ephemeral; for example, it was created on whiteboards or was personal, recorded in notebooks. It should also be noted that the technological environment was impoverished by today's standards: students did not have laptop computers or wireless access to the Internet (which did not exist as we know it, though the students did have e-mail). Ironically, this situation encouraged the use of physical objects for communication and compelled the students to gather in labs where computers were available for joint programming sessions, both of which would become important features of more recent agile programming methods.

35. Bill Curtis, "By the Way, Did Anyone Study Any Real Programmers?," in *Empirical Studies of Programmers*, ed. Elliott Soloway and Sitharama Iyengar (Norwood, NJ: Ablex, 1986), 256–62.

36. Diane B. Walz, Joyce J. Elam, and Bill Curtis, "Inside a Software Design Team: Knowledge Acquisition, Sharing, and Integration," *Communications of the Association for Computing Machinery* 36, no. 10 (1993): 62–77; also Curtis, "By the Way."

37. Holland, Reeves, and Larme, "Constitution of Intellectual Work"; Holland and Reeves, "Creativity and Rationalizability"; and Holland and Reeves, "Activity Theory and the View from Somewhere: Team Perspectives on the Intellectual Work of Programming," in *Contexts and Consciousness: Activity Theory and Human Computer Interaction*, ed. Bonnie Nardi (Cambridge, MA: MIT Press, 1994), 257–81.

38. The leading position of the laboratory in question made the programmers certain that this was the case.

39. All four members of the team are still involved in computer programming in some way: one is an academic, one is a researcher, one is a principal in a software company, and one is an independent programmer.

40. Brooks, *Mythical Man-Month*, 20.

41. Compare Lamb, *Software Engineering*, 35.

42. Holland, Reeves, and Larme, "Constitution of Intellectual Work."

43. Holland and Reeves, "Activity Theory." This free ideological space might be conceived in terms of Vygotsky's "zone of proximal development," the space between what learners are capable of by themselves and what they can accomplish with guidance. See James V. Wertsch, *Voices of the Mind: A Sociocultural Approach to Mediated Action* (Cambridge, MA: Harvard University Press, 1991). This is especially pertinent since the activities observed took place in a pedagogical setting. In the case of the programming team studied here, however, there was so little effort made to internalize the SE model, in fact, such obvious effort was made to resist some aspects of it, that it is clear that this space remained contested. The programmers had accumulated enough self-confidence in their own skills to allow themselves to resist the authoritarian discourse of SE and even to leverage themselves out of an obligation to pay serious attention to it.

44. Walz, Elam, and Curtis, "Inside a Software Design Team."

45. Ibid., 68.

46. Holland and Reeves, "Activity Theory."

47. Smith, "Collective Intelligence."

48. Many of these issues have nearly the status of religious conviction, so these kinds of discussions are not as trivial as they seem. A programmer's opinions about certain key texts in the field and certain programming practices will be taken by his peers as shorthand for a whole range of expectations. See Kelty's analysis of the world of open-source programming for many examples: Christopher Kelty, *Two Bits: The Cultural Significance of Free Software* (Durham, NC: Duke University Press, 2008).

49. This programmer's preferences are important in that they signal the beginning of a trend that would continue, in which programmers themselves began to prefer graphical user interfaces for their work.

50. This is one reason why programming is being rapidly adopted for other engineering tasks to provide for a means of simulating the task without actually carrying it out physically. Another reason is that it saves money, but given the leverage that the skill offers to those who have it, management may find that a mixed blessing.

51. Holland and Reeves, "Creativity and Rationalizability." For a more recent treatment of the distribution of agency through the activity of "cutting code," see Adrian Mackenzie, *Cutting Code: Software and Sociality* (New York: Peter Lang, 2006).

52. Brooks advises this: "Plan to throw one away; you will, anyhow" (*Mythical Man-Month,* 116).

53. The instructor's remark refers to a specific degree of computational complexity: NP-complete problems can be computationally solved, but not in any reasonable length of time for large datasets. For a discussion of the problem, see Michael P. Garey and David S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness* (San Francisco: W. H. Freeman, 1979).

54. Turkle, *Second Self.* See Pickering, *Mangle of Practice,* for the notions of the performativity of science and the material agency captured in human-engineered artifacts. Programming practice can amount to a fluid dialectic between programmer and program in the environment of machine and programming support software, reflecting Pickering's "mangle" of resistance and accommodation.

55. Holland and Reeves, "Creativity and Rationalizability."

56. Interestingly, a recent article observed: "Within Microsoft, we have found that when more people work on a binary, it has more failures" (Christian Bird, Nachiappan Nagappan, Brendan Murphy, Harald Gall, and Premkumar Devanbu, "Don't Touch My Code! Examining the Effects of Ownership on Software Quality," in *Proceedings of the 8th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on Foundations of Software Engineering* [2011]). The authors suggested that this fact came from the owner's better understanding of her own code and the notorious problem of communicating on programming teams. Their proposed solution is to experiment with changes in practice to include having "owners" of components do changes suggested by others. This study was not carried out through observation of programmers at work but entirely through examination of code fragment authors on a specific program in a versioning repository, compared with defect reports for that same program.

57. Naur, "Programming as Theory Building," 47–48.

58. See Christiane Floyd, Heinz Zullighoven, Reinhard Budde, and Reinhard Keil-Slawik, eds., *Software Development and Reality Construction* (Berlin: Springer-Verlag, 1992); and Yvonne Dittrich, Christiane Floyd, and Ralf Klischewski, eds., *Social Thinking—Software Practice* (Cambridge, MA: MIT Press, 2002).

59. Karen Holtzblatt and Hugh Beyer, "Making Customer-Centered Design Work for Teams," *Communications of the Association for Computing Machinery* 36, no. 10 (1993): 92–103.

60. A convenient source for the Agile Manifesto is at http://agilemanifesto.org/.

61. Two significant guides to varieties of agile programming practice are Ken Schwaber and Mike Beedle, *Agile Software Development with Scrum* (New York: Prentice-Hall, 2001); and Kent Beck and Cynthia Andres, *Extreme Programming Explained: Embrace Change* (Boston: Addison-Wesley, 2005).

Copyright of Information & Culture is the property of University of Texas Press and its content may not be copied or emailed to multiple sites or posted to a listserv without the copyright holder's express written permission. However, users may print, download, or email articles for individual use.

www.manaraa.com