

# **Programming a New Society**

**Modularity as an Instrument of Cooperation and Programmer Autonomy**

**from the 1960s to the Free Software Movement**

**Richard Williams**

**History Honors Thesis**

**April 12, 2013**

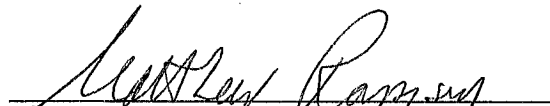
Programming a New Society: Modularity as an Instrument of Cooperation and Programmer  
Autonomy from the 1960s to the Free Software Movement

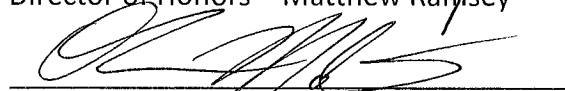
By  
Richard Williams

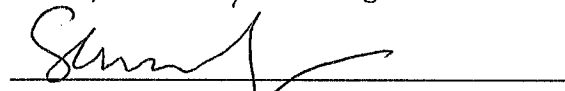
Thesis  
Submitted to the Faculty of the  
Department of History of Vanderbilt University  
In partial fulfillment of the requirements  
For Honors in History

April 2013

On the basis of this thesis defended by the candidate on April 24, 2013  
we, the undersigned, recommend that the candidate be awarded highest honors  
in History.

  
\_\_\_\_\_  
Director of Honors – Matthew Ramsey

  
\_\_\_\_\_  
Faculty Adviser – Ole Molvig

  
\_\_\_\_\_  
Third Reader – Sarah Igo

## Table of Contents

Introduction.....	3
The Computer Scientists behind Modular Programming.....	14
Modularity and Software Engineering: Programming’s Professional Aspirations.....	15
Clashing Conceptions of Modularity.....	20
Computer Scientists and New Communalists.....	27
Embracing Modularity, Facilitating Creativity, and Challenging Property.....	35
Attitudes toward Modularity.....	36
Modularity and the Art of Programming.....	41
Reactions to Proprietary Restrictions on Software.....	47
Free, Open Source, and Modular Software: The Rise of Cooperative Development.....	58
The Role of Modular Programming in a Distributed Development Model.....	59
Pragmatism and the Permissibility of Proprietary Software.....	67
Revolutionaries, by Intent and by Accident.....	72
Conclusion.....	81
Bibliography.....	85

## Introduction

Much has been made of the tendency of technological developments, particularly those associated with industrialism, to displace, deskill, and subjugate the people whose work they purportedly make more efficient. This is especially true of the electronic digital computer, which, since its invention during the 1940s, has been applied extensively in automating tasks that had previously been the responsibility of human workers. Even as computers devalued the skills of many workers, they also created a new category of skilled workers responsible for controlling them: computer programmers. Initially, the arcane nature of these programmers' expertise allowed them to demand a great deal of autonomy from their employers and managers. By the late 1960s, though, a software development methodology with the potential to change this had emerged: "modular" programming, or the decomposition of development tasks into smaller pieces that could be distributed amongst many programmers. Much of the surrounding scholarly literature emphasizes the role managers played in introducing modular programming, arguing that it represented an attempt to deskill programmers just as computers themselves had deskilled other workers. Yet modular programming was first formulated not by managers, but by programmers. These programmers believed that modular programming could provide the common set of standards programming needed to achieve the status of a profession. Working against the countercultural backdrop of the late 1960s and early 1970s, they hoped that by professionalizing, programmers would collectively secure their autonomy against managerial efforts to undermine it. Though their efforts to transform programming into a profession were largely unsuccessful, their efforts to convince other programmers of modularity's value were not. By the early 1980s, programmers in general had overwhelmingly accepted modularity as a worthy goal in programming, identifying modular programs as high-quality, aesthetically-

pleasing works and, in some cases, coming to consider the production of such programs a creative, artistic endeavor. When proprietary software firms began to leverage changes in intellectual property law to impose increasingly-strong restrictions on the acts of reading, sharing, and adapting programs, a subset of programmers, seeing these firms' actions as a threat to the values their conception of programming as a creative act had instilled, resisted. Forming what would come to be known as the free and open source software movement, they applied modular programming to distribute the task of producing freely-distributable replacements for proprietary software over many programmers, setting an example of something that would become a hallmark of Internet culture: the voluntary cooperation of enormous numbers of people on projects far beyond the capabilities of any one person. The managers of the 1960s may have hoped that modular programming would provide an instrument by which they could control programmers. However, much as computers and computer networks themselves came to be applied for purposes beyond the automation of human work—from the free dissemination of information to the coordination of revolutionary political movements—so did programmers come to apply modular programming to assert, exercise, and preserve their autonomy.

\* \*

\* \*

\* \*

“Information wants to be free, because the cost of getting it out is getting lower and lower all the time,” veteran of the late 1960s American counterculture Stewart Brand remarked at the Marin County, California Hackers' Conference of 1984, thereby unintentionally coining a phrase that would become a favored slogan among critics of intellectual property and other limitations on access to information. Four hundred programmers who identified as “hackers” were invited to the conference, and the roughly one hundred and fifty that actually attended included such prominent figures as Richard Stallman of the GNU Project and Steve Wozniak of Apple. Brand,

along with other members of the team responsible for producing the *Whole Earth Catalog*, had organized the conference for the purpose of discussing such contemporary trends in the computer industry as the expansion of intellectual property restrictions in the field of software development. According to Fred Turner's *From Counterculture to Cyberculture: Stewart Brand, the Whole Earth Network, and the Rise of Digital Utopianism*, Brand hoped that the conference would provide an opportunity to determine whether or not the act of hacking—defined not as the act of finding and exploiting weaknesses in computer systems but as the act of programming motivated by a sense of playfulness and a love of programming itself—might be a “precursor to a larger culture,” with hackers forming “the sort of cultural vanguard for the 1980s that the back-to-the-land and ecology crowds had hoped to be for the 1970s.”<sup>1</sup>

Turner's investigations of this conference and of other interactions between the segment of the counterculture Brand represented and the software development community together constitute an attempt to answer a question. How was it that, by the mid-1990s, “pundits, scholars, and investors alike saw the image of an ideal society: decentralized, egalitarian, harmonious, and free” in the “shiny array of interlinked devices” that characterized the arrival of “ubiquitous networked computing,” when, just thirty years earlier, computers had seemed to many student activists to be “the tools and emblems of the same unfeeling industrial-era social machine whose collapse they now seemed ready to bring about”? Turner argues that while the fact that this shift was possible at all may be attributed to a shift in the technology itself—specifically, to the increasing capabilities and decreasing size of computer hardware, culminating in the emergence of “personal” computers—the facts that the “particular utopian visions” that emerged were so strikingly similar to those of a particular subset of the counterculture, whom Turner termed the

---

<sup>1</sup> Fred Turner, *From Counterculture to Cyberculture: Stewart Brand, the Whole Earth Network, and the Rise of Digital Utopianism* (Chicago: University of Chicago Press, 2006), 1.

“New Communalists,” and that this subset interacted so often with the burgeoning community computer programmers through “network forums” such as Brand’s *Whole Earth Catalog* and its short-lived, software-focused extensions, the *Whole Earth Software Catalog* and the *Whole Earth Software Review*, indicate that the “digital utopianism” of the mid-1990s was descended directly from New Communalists’ idealism.<sup>2</sup>

Like others in the counterculture, the New Communalists distinguished themselves from the New Left by concerning themselves first and foremost with maintaining their autonomy, turning “inward, toward questions of consciousness and personal intimacy, and toward small-scale tools such as LSD or rock music as ways to enhance both” rather than “outward, toward political action.” For the New Communalists, though, “the work of expanding consciousness and increasing interpersonal intimacy was not an end in itself; it was a means by which to build alternative, egalitarian communities,” to the point that “in the early 1970s, some 750,000 people lived in ten thousand communes nationwide.” It was the members of these communes at whom the *Whole Earth Catalog* was aimed. Functioning “primarily as an evaluation and access device” with the purpose of letting the user “know better what is worth getting and where and how to do the getting,” the *Catalog* served to inform New Communalists seeking to get “back to basics” what those “basics” were. That is, it served to provide them with the knowledge both of what tools they could use to build successful communes and of how to access those tools. The way in which the *Catalog* gathered this information—by soliciting contributions from its readers—prefigured, Turner argues, the proliferation of user-generated content on computer networks such as the Internet, cultivating attitudes that viewed this sort of collaborative production of information as fundamentally tied to the New Communalists’ vision of a decentralized, nonhierarchical society and “would go on to shape popular attitudes toward networked

---

<sup>2</sup> *Ibid.*, 2-5, 136.

computing in the 1990s.” Moreover, the *Catalog*’s success was indicative of the New Communalists’ preoccupation “with reclaiming the products of government and industry and transforming them into ‘tools.’” Even though the “back-to-the-land” drive animating many of the *Catalog*’s readers often implied a certain degree of suspicion of the potential of these “products of government and industry” to control and oppress, their very interest in the *Catalog* implied the belief that such technologies could also be used to liberate, often by facilitating alternative, egalitarian forms of social organization.<sup>3</sup>

It was against this cultural backdrop that, in 1969, a group of programmers working for AT&T’s research-and-development subsidiary, Bell Laboratories, began work on a new kind of operating system. At least initially, their product, which they would dub “Unix,” was a creature of their own initiative. Having grown out of these programmers’ involvement in Bell Labs’ abandoned collaboration with General Electric and the Massachusetts Institute of Technology on another operating system called “Multics,” it received no funding from Bell Labs itself until the second year of development. Dennis Ritchie, one of the principal programmers of Unix, expressed sentiments strikingly reminiscent of the sort of preoccupation with the use of tools to facilitate the construction of communities that Turner argues characterized the New Communalists when he remarked on the intent driving the creation of Unix. This intent, he said, was to provide “not just a good environment in which to do programming, but a system around which a fellowship could form,” as he “knew from experience that the essence of communal computing [...] is not just to type programs into a terminal instead of a keypunch, but to encourage close communication.”<sup>4</sup> Unix’s significance went beyond its status as both a product of programmers’ agency and a tool for enhancing programmers’ sense of community, though. It

---

<sup>3</sup> *Ibid.*, 31-32, 79, 91.

<sup>4</sup> Dennis Ritchie, “The Evolution of the Unix Time-Sharing System,” in *Lecture Notes in Computer Science #79: Language Design and Programming Methodology* (Springer-Verlag, 1980).



also featured a number of technological innovations, among the most striking of which was the fundamentally modular nature of its design. Rather than being composed entirely of one monolithic program that provided all of the functionality expected of an operating system, Unix was divided into many small programs. One of these, the “kernel,” provided a set of core features to which every program was expected to require access, including starting or stopping other programs, managing the file system, and scheduling access to limited resources. The others, known as “utilities,” provided the remainder of Unix’s functionality independently of one another.

This design decision did not occur in a vacuum. Rather, it reflected a major shift that had engulfed the software development industry during the late 1960s: the formalization and propagation of modular programming techniques, which allowed development processes to be broken down into smaller pieces and distributed amongst many programmers. Unix itself did not fully take advantage of this, at least during its initial development. A relatively small group of programmers wrote the code for the operating system, with many of them writing multiple utility programs while also contributing to the code for the kernel. Over a decade and a half later, though, another operating system, explicitly intended to be “Unix-like” and to be compatible with programs written to run on Unix, did: GNU, whose name was a recursive acronym standing for “GNU’s Not Unix.” Developing GNU was the first task of the GNU Project and the Free Software Foundation, organizations founded by the programmer Richard Stallman to facilitate the development of a non-proprietary alternative to Unix (i.e., GNU) and, later, “free” software in general, with “free” software defined as software that users would be free to modify and redistribute at will. Motivated by the belief that contemporary expansions of intellectual property law as it applied to software represented threats to programmers’ autonomy and to the “hacker

ethic,” which regarded the sharing of code among programmers as a moral imperative, GNU, as well as later free and open-source software projects, relied on a development model that reasserted both of these on a massive scale, calling on large numbers of geographically-separated programmers to collaborate, with the work of each being incorporated into the whole. The modularity of Unix, as Stallman himself would note, made it easy for the GNU versions of each Unix utility to be implemented independently and, ultimately, combined to form a complete operating system. It thus played a key role in allowing the GNU Project to successfully apply this development model.

Dennis Ritchie, along with the other researchers behind the development of Unix, took the initiative in applying modular programming techniques to the development of an operating system whose express goal was to be something around which a “fellowship” could form. Richard Stallman and the free software movement would later make use of it in their attempts to assert their autonomy in the face of increasingly stringent restrictions on the reuse and redistribution of code. Yet the relatively sparse literature examining the causes and effects of the software development industry’s adoption of modularity as a basic methodological principle pays little attention to the role of programmers in formulating and propagating modular programming techniques. Instead, it focuses on the role of management in the introduction of these techniques and emphasizes the notion that they were intended to “routinize” computer programming, making it more like more conventional industrial work. For instance, the sociologist Philip Kraft's *Programmers and Managers* advances the argument that modular programming (though Kraft uses the term “structured programming,” which has come to denote a particular technique associated with modular programming) primarily served to deskill, and thus extend managers' control over, programmers. In it, Kraft claims that modularity “freed

managers from dependence on individual high-level software workers” and “made possible for the first time a genuine job-based fragmentation of labor in programming,” becoming “the software manager’s answer to the assembly line.” Kraft acknowledges that the Taylorist division between labor of the “head” and labor of the “hand” that he claims had emerged in the form of a separation between low-level “coders,” mid-level “programmers,” and high-level “systems analysts” was somewhat tenuous since “everyone does at least some coding, for example.” However, he nevertheless insists that the industry was characterized by an increasing tendency to divide workers along these lines. To his credit, he explicitly notes one of the major omissions of his study in his introduction: he chose to avoid examining the role of academic and research-oriented programmers, or “computer scientists,” in formulating modular programming, instead focusing on “what managers have selected from the work of researchers in order to further their own ends.” He does not, however, note the other major omission: he never acknowledges that programmers might have actually influenced the purposes to which modular programming techniques would be applied, rather than simply playing a role in inventing those techniques.<sup>5</sup>

Chris Benner’s examination of the manner in which workers in Silicon Valley’s information technology sectors have organized further shows the insufficiency of Kraft’s approach for understanding the impact of modular programming on programmers. Benner argues that, rather than taking the form of the more-familiar industrial unions that dominated the labor movement in the twentieth century, collective associations amongst these workers have tended to eschew the term “union” in favor of “guild” and to take forms more analogous to professional associations, benefiting their members primarily by having “improved their member’s career opportunities, through improving skill development, facilitating access to new job opportunities,

---

<sup>5</sup> Philip Kraft, *Programmers and Managers: The Routinization of Computer Programming in the United States* (New York: Springer-Verlag, 1977), 9, 15-16, 58-59.

and organizing advocacy efforts,” not through collective bargaining. These guilds developed, he claims, around occupations in which “employment conditions change rapidly over time, with workers being more connected with their occupation and trade, rather than a particular employer.” That such a form of organization would have predominated within the information technology industry implies that this industry remains one in which individual craft, after the manner of artisans rather than that of industrial workers, is of foremost importance. However, it does not explain why this is the case. If the 1960s shift in approach to software development really was more an attempt to deskill and proletarianize programmers than anything else, why did it, apparently, fail?<sup>6</sup>

It may well be the case that managers, as Kraft argues, deliberately encouraged programmers to make use of modular programming techniques, hoping that their use had the potential to displace, deskill, and subjugate these programmers. However, modular programming techniques did not originate in the minds of managers, and the computer scientists who first articulated these techniques during the 1960s and early 1970s did not share the managers’ interest in making programmers easier to control. Formulated and propagated against a countercultural backdrop infused with the New Communalists’ preoccupation with transforming the potentially-oppressive technologies produced by industry and government into tools for liberation—and amongst a community of programmers who, as Fred Turner argues, had substantial interaction with representatives of precisely that countercultural current in the form of Stewart Brand and the *Whole Earth Catalog*—modular programming techniques played the role of such tools both in intent and in practice. The researchers who formulated them did so intending for them to provide programming with a base of practical knowledge, driven by the

---

<sup>6</sup> Chris Benner, “Computers in the Wild!: Guilds and Next-Generation Unionism in the Information Revolution,” in *Uncovering Labour in Information Revolutions, 1750–2000*, ed. Aad Blok and Greg Downey (Cambridge: Cambridge University Press, 2003), 182, 190.

conviction that to do so would lead to the advancement of the field of computer programming and, perhaps, would enable programmers to gain recognition as professionals. Thus, for these researchers, modular programming was a tool by which programmers could, collectively, secure their autonomy. By the late 1970s and early 1980s, programmers had overwhelmingly adopted modularity as a goal to which to aspire in software development, and were considering these techniques as a means through which they would be able to exercise and share the products of their creative energies, much as the New Communalists made use of small-scale tools in artistic endeavors designed to enhance both individual consciousness and personal intimacy among individuals. Finally, when developments in intellectual property law began to impose increasingly-restrictive constraints on behavior (such as the sharing of source code) that had previously been common among programmers, they undercut the ability of programmers to use modularity for this purpose. In response, programmers, beginning in the mid 1980s with the GNU Project, undertook a series of enormous, cooperative projects, all made possible by the use of modular programming techniques, the fruits of which would be freely available to all programmers and, in many cases, would replicate features provided by newly-proprietary code. Here, then, modular programming served as a tool for the creation of networked, collaborative communities of programmers reminiscent of both the New Communalists' communes and the *Whole Earth Catalog* itself.

Viewing modular programming through this lens renders the apparent failure of the managers' attempt to industrialize the industry less surprising by making modular programming more than just a means by which managers attempted to subjugate programmers. It makes it, in both intent and application, a means by which programmers autonomously acted to improve both their own lot and the lot of their fellows. From modular programming's origin as an attempt to

provide a practical foundation for the field of programming to its use as a means by which programmers could share the products of their creativity up through its employment by networks of programmers collaborating to build free and open source software for all to use, modular programming affirmed the New Communalists' belief in the possibility for technology—even technology with an apparent potential to act as an instrument of control—to serve as a tool of liberation.

## The Computer Scientists behind Modular Programming

In October 1968, the same year Stewart Brand published the first edition of the *Whole Earth Catalog*, the NATO Science Committee invited “50 experts from all areas concerned with software problems” to Garmisch, Germany for the first NATO Conference on Software Engineering. Intended “to shed further light on the many current problems in software engineering” and “to discuss possible techniques [...] which might lead to their solution,” the conference aimed to address one problem in particular: large software projects’ perceived tendency to be excessively error-prone and inflexible. As the capabilities of computer hardware had grown, the size, complexity, and importance of the software programmers were expected to produce had grown as well, “placing demands on us which are beyond our capabilities [...] at this time.” Central to the proposed solutions to this problem were the techniques that would come to be associated with modular programming. The computer scientists who formulated these techniques hoped that they would provide the theoretical and practical framework necessary for programming to attain the status of a profession in the form of “software engineering.” Their preferred approaches to making programs modular diverged from those of managers to a degree sufficient to set their efforts to professionalize programming at odds with managerial efforts to the same end. Finally, their efforts were linked to the New Communalists’ efforts to transform society not only by a common time period and even common participants, but also by similar motivations—where the New Communalists hoped to assert their autonomy and to protect the world from the threat of nuclear weapons, these computer scientists hoped to secure programmers’ autonomy and to protect society from the potential dangers of poorly-written software.<sup>7</sup>

---

<sup>7</sup> Peter Naur and Brian Randell, *Software Engineering: Report on a conference sponsored by the NATO Science Committee, Garmisch, Germany, 7th to 11th October 1968* (Brussels: NATO Scientific Affairs Division, 1969),

## **Modularity and Software Engineering: Programming's Professional Aspirations**

“The phrase ‘software engineering,’” wrote the computer scientists Peter Naur and Brian Randell in their report on the Garmisch conference, “was deliberately chosen” to describe the conference’s subject because of its “provocative” implication that there existed a “need for software manufacture to be based on the types of theoretical foundations and practical disciplines, that are traditional in the established branches of engineering.”<sup>8</sup> Even before the conference had begun, it was believed—by the organizers of the conference, if not necessarily by all of the attendees—that the solution to the perceived software crisis would involve the transformation of programming into a discipline similar to the established branches of engineering. It would involve, in other words, the professionalization of programming.

“Professionalism” is a problematic term, partially because it carries with it a set of meanings and connotations that make it difficult to define in a manner that is both satisfactory and concise. In *Professionalism: The Third Logic*, the sociologist Eliot Freidson describes the ideal world associated with professionalism as one in which “those workers who have the specialized knowledge that allows them to provide especially important services have the power to organize and control their own work,” protected by legal guarantees that “only they can offer their particular services to consumers or hold jobs performing them in organizations: neither consumers nor managers are free to employ anyone else” and that “only members of the occupation have the right to supervise and correct the work of colleagues.” All this comes with the caveat that the members of the profession “do not abuse these exclusive rights,” if only because “they are more dedicated to doing good work for their own satisfaction and for the benefit of others than to maximizing their income.” From this description, another reason the

---

13-14, 17.

<sup>8</sup> *Ibid.*, 13.



term is problematic becomes apparent: as Friedson acknowledges, this ideal professional world has never existed. Rather, “where some of [its] elements have existed, predicted virtues are always accompanied by unanticipated vices,” with some members of professions putting “economic advantage ahead of the good of their clients,” so that the world in question represents an “ideal type” or, more plainly, a “pipe-dream.”<sup>9</sup>

A key aspect of professionalism as Friedson describes it is that members of a profession alone have the right to “supervise and correct the work of colleagues.” For them to be able to do so, though, there must be some notion of what constitutes correct work. There must exist, to borrow Naur and Randell’s words, “theoretical foundations and practical disciplines” on which that work will be based and by which that work can be judged. Ensmenger argues that programmers had attempted to professionalize in the early 1960s and that, as part of this enterprise, they had formed organizations such as the Association for Computing Machinery (ACM). This academically-oriented organization was devoted to the study of computer science and, in particular, to a strategy of professionalization that focused on providing just the sort of theoretical and practical foundations for the aspiring profession of programming to which Naur and Randell would later refer when explaining the choice of the phrase “software engineering.” Yet Ensmenger’s treatment of the Garmisch conference and of the use of the term “software engineering” (and, by extension, the modular programming techniques developed to facilitate software engineering) explicitly separates both from the issue of programmers’ attempts to professionalize, instead describing them as attempts to reassert “control over a recalcitrant workforce.” The literature surrounding the genesis of modular programming and of software engineering suggests that this separation is not justified. It reflects both the ethical concerns and

---

<sup>9</sup> Friedson, Eliot, *Professionalism: The Third Logic* (Cambridge: Polity Press, 2001), 1-2.

the need for standards on which to base and by which to judge work that are associated with professionalism.<sup>10</sup>

Consider, for example, Douglas McIlroy's keynote address from the Garmisch conference, "Mass Produced Software Components." McIlroy was, in many ways, the archetypal computer scientist. Not only did he have a Ph. D. in applied mathematics from Cornell University, he was also the head of the Computing Techniques Research Department at AT&T's Bell Laboratories and would eventually write a number of the tools associated with Unix, of which the GNU Project would later produce non-proprietary replacements. That McIlroy was arguing for the use of modular programming seems clear from the example he uses to introduce his argument, which implies that software projects ought to be broken down into modules (or, to use his term, components) that can then be reused: "when we undertake to write a compiler, we begin by saying 'what table mechanism shall we build?' Not, 'What mechanism shall we use?' [...] I claim we have done enough of this to start taking such things off the shelf." That he was advancing it in a manner reminiscent of the managers' purported attempts to use it control programmers also seems clear, this time from his statements that he "would like to investigate the prospects for mass-production techniques in software" and that what he was proposing was "simply industrialism, with programming terms substituted for some of the more mechanically oriented terms appropriate to mass production." His motivations, however, diverged sharply from those attributed to the managers. His opening claim that "[w]e undoubtedly produce software by backward techniques" and "get the short end of the stick in confrontations with hardware people because they are the industrialists and we are the crofters" betrays an overriding concern with the idea that the field of software development was somehow behind the field of hardware development, a more traditional branch of engineering. This, taken together with

---

<sup>10</sup> Ensmenger, *The Computer Boys Take Over*, 174, 198.

McIlroy's reference to programmers in general with the pronoun “we” in statements such as “I don't think we are ready to make software subassemblies of that size on a production basis” and his insistence that the profit motive acted as a “prime hangup” of manufacturers with regard to their ability to produce “high class” component libraries “of general utility,” indicates that he identified primarily with programmers and was interested in providing a common base of knowledge—high class component libraries—on which programmers could build. The same is true of his reasons for regarding government as a good choice for initial funding: that his proposals would result in “across-the-board improvement in systems development.” Finally, his claim that “the software industry is weakly founded” due in part to the lack of a software components industry suggests that he was interested in enhancing the foundation of software industry and, thus, providing a practical basis for the establishment of programming as a profession.<sup>11</sup>

The prominent Dutch computer scientist Edsger Dijkstra was likewise a vocal advocate of modular programming. In particular, he famously condemned the use of “go to” statements, or commands in a program that cause the flow of execution to “jump” to some specified point in a program when executed, in his paper, “Go To Statement Considered Harmful,” advocating that they be replaced with control structures such as loops and procedure calls. The systematic use of such structures, or “structured programming,” was one of the chief techniques used to make program modular. Upon receiving the ACM's Turing Award in 1972, Dijkstra gave an address entitled “The Humble Programmer.” In it, he related the story of “a turning point in [his] life” in which, early in his career, he had chosen to pursue programming rather than theoretical physics on the grounds that “up till that moment there was not much of a programming discipline,” but that “automatic computers were here to stay” and he might thus “be one of the persons called to

---

<sup>11</sup> Naur and Randell, *Software Engineering*, 138-139, 144, 148, 150.

make programming a respectable discipline in the years to come.” The beginning of his speech sheds light on his use of the term “respectable discipline.” When introducing his dilemma, he asked, “But was [programming] a respectable profession? After all, what was programming? Where was the sound body of knowledge that could support it as an intellectually respectable discipline?” In a manner strikingly similar to that of McIlroy, he then proceeded to contrast the intellectual foundation of software development with that of hardware development, saying, “I remember quite vividly how I envied my hardware colleagues, who, when asked about their professional competence, could at least point out that they knew everything about vacuum tubes, amplifiers and the rest, whereas I felt that, when faced with that question, I would stand empty-handed.” Dijkstra’s very reason for becoming a programmer in the first place, and likely for advocating the use of structured programming, was to contribute to the professionalization of programming, to contribute to a body of knowledge that, in his view, was too small to serve as the basis for a profession at the time he became a programmer.<sup>12</sup>

The aspects of professionalism associated with the rise of “software engineering” and of modular programming techniques are not limited to the creation of an adequate foundation for programming as a profession. Among the first subsections of the Naur and Randell report was one that considers the impact of software development practices on society and, as such, was aptly entitled “Software Engineering and Society.” The existence of this subsection, according to Naur and Randell, can be attributed to the fact that “[o]ne of the major motivations for the organizing of the conference was an awareness of the rapidly increasing importance of computer software systems in many activities of society.” That this section was featured so prominently in the report indicates either precisely the sort of preoccupation with the impact of software

---

<sup>12</sup> Edsger Dijkstra, “The Humble Programmer,” *Communications of the ACM* 15, no. 10 (1972): 859-860, doi:10.1145/355604.361591.

developers' work on others that Freidson attributed to professionals as an ideal type, or, at least, indicates a desire for software developers to be perceived as having this sort of preoccupation and, therefore, as embodying professionalism. Similarly, Brian Randell himself expressed sentiments embodying such preoccupation during a discussion of the implications of the existence of a "software crisis," saying, "There are of course many good systems, but are any of these good enough to have human life tied on-line to them, in the sense that if they fail for more than a few seconds, there is a fair chance of one or more people being killed?"<sup>13</sup>

Modular programming, at least in the eyes of the computer scientists of the 1960s who articulated its importance, was the product of a collective effort to create a shared set of standards and body of knowledge that would contribute to the establishment of programming as a "respectable discipline." By providing programmers with a theoretical and practical foundation according to which they could supervise and correct each other's work, modular programming would allow the community of programmers to obtain one of the qualities Eliot Freidson describes as characteristic of a profession as an ideal type: self-regulation. This, together with the evidence of another of these qualities, concern for the social good, in the computer science literature of the late 1960s, indicates that these computer scientists' research into modular programming represented part of programmers' ongoing attempts to professionalize. Drawing conclusions about their reasons for wanting to transform programming into the profession of software engineering necessitates a more careful exploration of the literature they produced.

### **Clashing Conceptions of Modularity**

The computer scientists who formulated the techniques associated with modular programming and the related concept of software engineering appear to have had the establishment of programming as a profession as their first concern. However, this does not

---

<sup>13</sup> Naur and Randell, *Software Engineering*, 120.

necessarily imply that they were acting primarily with the interests of programmers, rather than those of managers, in mind. After all, as Ensmenger points out, managers may have viewed professionalism, provided it stayed “corporate-friendly,” as a way to simplify the task of measuring the qualifications of potential programmer employees and to reduce their own dependency on individual programmers by standardizing the practice of programming.<sup>14</sup> Examining more closely what the computer scientists in question meant by “modular programming,” and, in particular, the relationship between their conceptions of what modularity was and how it was to be applied and those of managers reveals a key subject on which the managerial literature differed from that produced by computer scientists: the use of modular programming techniques to promote a Taylorist division of labor.

Before proceeding to a discussion of managers’ conceptions of modularity, it is necessary to more precisely specify the meaning of the term “manager,” and to deal in particular with the extent to which programmers and managers overlapped. Certainly it was possible for some programmers to be in charge of other programmers and thus to act as managers in some sense. However, Ensmenger’s use of the term “managers” appears to apply primarily to those individuals whose training and experience deals with the organization and supervision of employees rather than the development of software. Moreover, Ensmenger argues that, particularly when compared with workers in the traditional engineering disciplines, programmers had relatively few opportunities to advance up the “corporate ladder” into management positions, perhaps due to the fact that it “was just not clear to many corporate employers how the skills—and personality types—possessed by programmers would map onto the skills required for management.”<sup>15</sup> Ensmenger claims that situation is part of what led to programmers wishing to

---

<sup>14</sup> Ensmenger, *The Computer Boys Take Over*, 168.

<sup>15</sup> *Ibid.*, 22, 166.

establish their occupation as a profession in the first place, as a way of compensating for the lack of opportunities to move into management. Likewise, it led to managers seeking to promote a division of programmers' labor along Taylorist lines. Since programmers were perceived as lacking the skills needed to effectively manage themselves and each other, doing so would purportedly enable large software projects to be completed more reliably and efficiently by making it easier for managers to control programmers.

As Philip Kraft describes it, such a division of labor involves a process whereby “engineers broke down a product’s manufacture into the smallest possible component parts” with the result that “employers needed fewer skilled production workers.”<sup>16</sup> That this process resembles the process of breaking up a program into modules that can be implemented more-or-less independently is clear. Brian Rothery’s guidebook for managers, *Installing and Managing a Computer*, described how a successful manager of programmers must divide the task at hand into “simple work units,” that is, modules, to be implemented by “simple programmers,” lest said manager lose control of the task and thus be “held in contempt by clever programmers dangerously maintaining control on his behalf.”<sup>17</sup> Thus, according to the managerial conception of modular programming, all important design decisions would be made by the managers, who would decompose the task into modules and allocate each module to a programmer to implement without having any knowledge of, or control over, the overall design of the system.

This contrasts with the approach outlined in a paper written by the Canadian software engineering pioneer David Parnas, who was notable for developing the notion of “information hiding” or of ensuring that certain pieces of data can only be accessed by certain pieces of code. As its name, “On the Criteria To Be Used in Decomposing Systems into Modules,” would

---

<sup>16</sup> Philip Kraft, *Programmers and Managers: The Routinization of Computer Programming in the United States* (New York: Springer-Verlag, 1977), 20.

<sup>17</sup> Brian Rothery, *Installing and Managing a Computer* (London: Business Books, 1968), 152.

suggest, this paper's purpose was to "suggest some criteria which can be used in decomposing a system into modules" and, more generally, to discuss the process of making a program modular. First, it is notable that, while enumerating various benefits of modular programming, Parnas explicitly claimed that one of these was "managerial—development time should be shortened because separate groups would work on each module with little need for communication." Initially, this appears to indicate a striking resemblance between Parnas' conception of how programs ought to be modularized and the aforementioned Taylorist conception of the same. Modularizations of a system would, following Parnas' method, be designed explicitly to reduce communication between the programmers working on each module, a process that could facilitate the ability of managers to keep the overall design of that system out of the control and the knowledge of the programmers of each module. However, considering the specific method Parnas proposed yields a very different interpretation. Parnas claimed that "it is almost always incorrect to begin the decomposition of a system into modules on the basis of a flowchart," since doing so—having each module represent one step in a process—could strongly couple each module to the modules representing the steps preceding and following its own step, so that changes to one module might well require changes to the other modules as well. Parnas instead proposed these alternative criteria for modularizing a system: "one begins with a list of difficult design decisions or design decisions which are likely to change. Each module is then designed to hide such a decision from the others. Since, in most cases, design decisions transcend time of execution, modules will not correspond to steps in the processing." In order to encapsulate specific design decisions in modules, though, the programmers writing those modules would need to be aware of those design decisions. Moreover, the fact that these encapsulated design decisions were, according to Parnas' criteria, explicitly the "difficult" ones and the ones that



were “likely to change” implies that these modules would not be, as Rothery put it, “simple work units” that could be implemented by “simple programmers.” They would demand cleverness on the part of the programmers implementing them rather than lending themselves to the deskilling of those programmers.<sup>18</sup>

Parnas was not the only exponent of modularity whose preferred approach to making programs modular contradicted the Taylorist one embodied by Rothery’s statements. Edsger Dijkstra, in his 1972 article “Notes on Structured Programming,” outlined an approach that did so just as clearly. In it, Dijkstra’s first concern was not with the manageability of programmers, but rather with programmers’ need to be confident of the correctness of their programs. Remarking that “it is fairly hopeless to establish the correctness [of programs] beyond even the mildest doubt by testing” due to the often astronomically-large number of cases that would need to be tested, he claimed that, instead, successful proofs of correctness would have to operate by “taking the structure of the mechanism into account.” Given that “it is not only the programmer’s task to produce a correct program but also to demonstrate its correctness in a convincing manner,” he argued, “the above remarks have a profound influence on the programmer’s behavior: the object he has to produce must be usefully structured,” that is, structured in a manner that would facilitate demonstrations of a program’s correctness. To this end, he suggested, programmers ought to arrange their programs in layers. The main body of the program—the top layer—would be implemented in terms of the second-highest layer, which would hide the details involved in the implementation of still-lower layers from the top layer and would thus allow “the correctness of the main program” to “be discussed and established” independently of that of the lower layers. The task of writing a program would thus be divided

---

<sup>18</sup> David Parnas, “On the Criteria To Be Used in Decomposing Systems into Modules,” *Communications of the ACM* 15, no. 12 (1972): 1053-1054, 1058, doi:10.1145/361598.361623.

into separate tasks, each “structurally similar to the first one,” involving the production of one layer. The end result of this process would be a program built as a set of distinct, hierarchically-organized levels of abstraction, each of which would provide the functionality needed to implement the level residing immediately above it in the hierarchy, implemented in terms of the level residing immediately below. The highest of these levels would be the main program, intended to provide precisely the functionality desired by the user, while the lowest would be the bare hardware. Each level of abstraction, like the top level, could be “understood all by itself” and could thus be proven correct independently of the other levels.<sup>19</sup>

This process might seem at first as though it would lend itself, even if unintentionally, to a concentration of all important design decisions into the hands of the people writing the main program (or even the specification for the main program) and a clear division of the task of programming into “simple work units” in the form of the separate layers of abstraction. However, consider the sorts of decisions that programmers would have to make while implementing each of these layers. To use Dijkstra’s words, in programming one such layer, “we have to decide upon data structures to provide for the state space of the upper [layer]; furthermore we have to make a bunch of algorithms, each of them providing an implementation of an instruction assumed” in the implementation of the upper layer. Far from being unimportant, the choice of a particular data structure needed to represent the data to be used by the layer immediately above or of a particular algorithm to implement a piece of functionality needed by that layer would have the potential to drastically impact the performance and stability of the program as a whole. Moreover, a large part of Dijkstra’s reasoning for the use of the model he described hinged precisely on its ability to avoid committing to any such design decisions until

---

<sup>19</sup> Edsger Dijkstra, “Notes on Structured Programming,” in *Structured Programming*, ed. Ole-Johan Dahl, Edsger Dijkstra, and C. A. R. Hoare (London: Academic Press Ltd., 1972), 5-6, 61-62.

as late as possible by pushing code amounting to such commitments as far “down” in the hierarchy of layers as possible. Doing so, he argued, would reduce the amount of code dependent on any particular design decision by reducing the number of layers shaped by that decision, thereby allowing programmers to change such a decision relatively easily. It was for this reason that he ended his article with a declaration of the virtue of seeing “[p]rogramming (or problem solving in general?) as the judicious postponement of decisions and commitments!” Yet this would not be the only effect of postponing decisions in this manner. Doing so would also ensure that the programmers working on lower layers of abstraction would both be aware of and have a degree of input in the design decisions that had been postponed until these levels. Thus, any attempt to apply Dijkstra’s model as Dijkstra himself envisioned it would undermine attempts to impose a straightforward, Taylorist division of the labor involved in the development of levels of abstraction into “head” labor associated with the highest level and “hand” labor associated with the others.<sup>20</sup>

The computer scientists responsible for the key techniques of modular programming conceived of several distinct processes according to which large programs could be decomposed into modules, often arguing for their proposed processes on the grounds that they would produce easily-adaptable programs. Where managerial conceptions of modular programming regarded it as a means to divide programs into simple work units that could be implemented by simple programmers, these computer scientists sought to decompose programming tasks along lines that made programmers responsible for important design decisions and made their tasks anything but simple. Rather than reinforcing managers’ attempts to assert their control over programmers, the application of modular programming techniques, at least as envisioned by their creators, tended to subvert them.

---

<sup>20</sup> *Ibid.*, 62, 84.

## Computer Scientists and New Communalists

If the computer scientists behind modular programming did not have managerial interests at heart, what did they hope to achieve by providing programming with the practical foundation it would need to professionalize? It would be a mistake to attribute the same motivations to all of these computer scientists. Nevertheless, two distinct threads driving the pursuit of professionalism in programming can be distinguished in the literature they produced: ethical concerns about the social implications of the increasingly-important role played by software and less-altruistic concerns for the benefits professionalism would have for programmers themselves. Somewhat paradoxically, given that these concerns drove computer scientists to pursue professionalism, both of these have parallels in the New Communalists' motivations for attempting to reclaim the products of government and industry and turn them into tools, to the point that modular programming might be regarded as a "tool" the computer scientists employed to pursue the same ends as the New Communalists.

The first of these two threads, the sense that an ethical imperative for programmers to professionalize existed, appeared in Brian Randell's aforementioned concern that poorly-designed systems shouldering important responsibilities might cause harm to actual people, embodied in his skepticism that "any [systems] [were] good enough to have human life tied on-line to them."<sup>21</sup> Randell expressed this concern in the midst of a discussion of the extent of the purported software crisis, and he was not alone. Edward E. David, Jr. and Alexander Fraser issued a joint statement claiming that "[p]articularly alarming is the seemingly unavoidable fallibility of large software, since a malfunction in an advanced hardware-software system can be a matter of life and death, not only for individuals, but also for vehicles carrying hundreds of

---

<sup>21</sup> Naur and Randell, *Software Engineering*, 120.

people and ultimately for nations as well.”<sup>22</sup> Both of these perspectives effectively impose upon programmers an ethical obligation to write code that would be considered “good,” which, at least in the aftermath of the Garmisch conference, would imply that said code be modular. Recent discussions of the question of whether or not programming ought to adopt such aspects of a profession as certification have echoed these sentiments. For example, in a 1998 article appearing in an ACM newsletter, Don Gotterbarn asserted that “[g]iven the degree of impact of our products, developing and maintaining quality products is a moral obligation,” that “[t]he impact of [Software Engineering] is too great to be used in frivolous and dangerous ways,” and that “[t]here is significant agreement on this among practitioners.”<sup>23</sup>

In the same discussion that gave rise to Parnas’ remarks, R.C. Hastings offered a contrasting viewpoint by saying that he was “very disturbed that an aura of gloom has fallen over this assembly” and that he did not “think software engineering should be confused with time sharing system engineering. Areas like traffic control, hospital patient monitoring, etc., are very explosive, but are very distinct from general purpose computing.”<sup>24</sup> He was joined by Alan Perlis, who noted that certain software projects “have taken a lot longer to develop than we would have wished” with “disappointing” results, but proposed that “perhaps we are exaggerating the importance of these facts. Is bad software that important to society? Are we too worried that society will lose its confidence in us?”<sup>25</sup> Hastings and Perlis both presented opinions that failed to impose the sort of ethical urgency on “good” software engineering practices that Randell, David, Fraser, and Gotterbarn’s perspectives did. However, Perlis’ last question implied a different justification for the use of such “good” practices, without the ethically-charged

---

<sup>22</sup> *Ibid.*

<sup>23</sup> Don Gotterbarn, “Software Engineering as a Profession,” *ACM SIGSOFT Software Engineering Notes* 23, no. 6 (1998): 206, doi:10.1145/291252.295145.

<sup>24</sup> Naur and Randell, *Software Engineering*, 120.

<sup>25</sup> *Ibid.*, 121.

character of the first: a self-interested justification that sought to prevent society from “losing its confidence” in programmers. The perception that, as the Soviet computer scientist Andrei Ershov remarked during his keynote address at the ACM’s Spring Joint Computer Conference of 1972, as “the claim of programmers to be a special employee has come to be disputed,” “authority over the freewheeling brotherhood of programmers is slipping into the paws of administrators and managers—who try to make the work of the programmers planned, measurable, uniform, and faceless” may explain this concern for maintaining society’s confidence in programmers.<sup>26</sup> This is consistent with Nathan Ensmenger’s claim that programmers who attempted to professionalize were driven as much by the desire for higher status, better pay, protection from the fluctuations of the labor market, and, particularly, a greater degree of autonomy, as they were by the desire to guarantee “basic standards of quality and reliability” to their clients.<sup>27</sup>

Both in their ethically-charged arguments for the social necessity of professionally-standardized programming practices and in their interest in preserving programmers’ autonomy, the reasons these computer scientists offered for professionalizing programming (and, therefore, for encouraging the use of modular programming techniques) mirrored at least some of the reasons that Fred Turner argues motivated the New Communalists in their use of “tools” derived from the technological products of mainstream society. The New Communalists “embraced the notion that small-scale technologies could transform the individual consciousness,” to the point that an early manifestation of New Communalist social ideals, the San Francisco Trips Festival of 1966, constituted a “techno-social hybrid,” which “surrounded dancers with the lights, images, and music of electronic media” so that “[t]o the extent that they felt a sense of communion with one another, the sensation was brought about by their integration into a single techno-biological

---

<sup>26</sup> Andrei Ershov, “Aesthetics and the Human Factor in Programming,” *Communications of the ACM* 15, no. 7 (1972): 502, doi:10.1145/361454.361458.

<sup>27</sup> Ensmenger, *The Computer Boys Take Over*, 166, 168.

system.” The New Communalists’ efforts represented their attempts to answer the two questions Turner identifies Stewart Brand’s generation as having faced as they came of age: “How could they keep the world from being destroyed by nuclear weapons or by the large-scale, hierarchical governmental and industrial bureaucracies that built and used them? And how could they assert and preserve their own holistic individuality in the face of [a world dominated by such bureaucracies]?” By turning the products of these bureaucracies into tools that they would use to build alternative, communal societies, they believed they would achieve the second of these goals, and by influencing the direction of mainstream society through the example of these alternative societies, they believed they would achieve the first. Similarly, the computer scientists of the late 1960s and early 1970s saw modular programming techniques as “tools” that would make programs easier to understand, debug, and maintain, thereby protecting society from the dangers posed by buggy, poorly-written programs responsible for controlling critical devices, and would provide a practical basis of knowledge for programmers, thereby allowing programmers to professionalize and maintain their autonomy in the face of perceived attempts by managers to “routinize” their work.<sup>28</sup>

The connections between the computer scientists’ modular programming techniques and the New Communalists’ “tools” were not limited to their similarities of purpose. For example, the idea of programming methodologies being “tools” in the sense that the New Communalists used the word would probably not seem strange to the New Communalists themselves. After all, “the great majority of the ‘tools’ offered by the *Whole Earth Catalog* were books and periodicals,” which, for example, might have “offered ways to [...] transform the products of the military-industrial complex, such as army jackets and boots, into individual statements of

---

<sup>28</sup> Turner, *From Counterculture to Cyberculture*, 43, 67, 74.

personal identity,” so that many of the *Catalog*’s tools were themselves informational, and particularly methodological, in nature.<sup>29</sup> The forms of autonomy that the computer scientists and the New Communalists championed were also limited in similar ways. Much as the professionalization of programmers would enhance the autonomy of a small and in some sense elite portion of society, so did the *Catalog* emphasize “that its products belonged to the do-it-yourself tradition of frontier elite” and that its readers “might be exceptional individuals, might be part of a vanguard” of people who would be the first to “merge consumption and technology with the dream of pre-industrial community.”<sup>30</sup> Just as there was a sense among the New Communalists that their own elite status was temporary and that the tools that made their alternative forms of social organization possible “would be deployed first by an elite and later by the whole population,” though, there was also a sense among computer scientists that, to again quote Andrei Ershov, “the highest aesthetic idea of our profession” was “to make the art of programming public property, and thereby to submerge our elite exclusiveness within a mature mankind,” so that the elite status of programmers would likewise be temporary and programmers would not long be “arbiters between the lay generality of mankind and the arcane informational model of the world hidden in the machine.”<sup>31</sup> Finally, there is good reason—beyond the similarities of purpose between the computer scientists’ formulation of modular programming and the New Communalists’ use of tools—to believe that the computer scientists in question would have been aware of, and in some cases directly influenced by, the New Communalists’ ideology. Alan Kay, the computer scientist who coined the term “object-oriented” for a particular modular programming paradigm and designed the programming language Smalltalk, had come across a copy of the *Whole Earth Catalog* in 1969. In 2004, he remarked that he and his

---

<sup>29</sup> *Ibid.*, 92.

<sup>30</sup> *Ibid.*, 93.

<sup>31</sup> Ershov, “Aesthetics and the Human Factor in Programming,” 505.



colleagues “thought of the *Whole Earth Catalog* as a print version of what the Internet was going to be” and drew some of their ideas from its pages; for instance, he remembered thinking that, just as the *Catalog* made it “easier to do your own composting,” “you should have the ability to deal with complicated ideas by making models of them on the computer.”<sup>32</sup>

It is likely that many motivations drove the computer scientists behind the techniques associated with modular programming in their attempts to use these techniques as a foundation for the establishment of software engineering as a profession. Yet the motivations that seem most prominent in the literature these computer scientists produced, the desire to secure the collective autonomy of programmers and the desire to prevent poorly-written software from causing harm to society, closely mirrored those of the contemporaneous New Communalists. Just as the New Communalists sought to harness the technological products of bureaucracy in constructing alternative communities as part of an effort both to secure their collective autonomy and to transform mainstream society into one that would not threaten the world with its use of nuclear weapons, these computer scientists sought to put modular programming to use in a way that would both secure programmers’ autonomy and prevent poorly-written software from causing harm to individuals or to society as a whole.

\* \* \* \* \*

The computer scientists of the 1960s were not successful in remaking the occupation of computer programming into the profession of software engineering, at least as far as society was concerned. As Ensmenger notes, in 1967, the US Civil Service Commission declared data processing personnel to be nonexempt employees, and, therefore, to be non-professional technicians in the eyes of the government. This is not because they failed to provide the practical basis for such a profession, or, for that matter, because programming failed to acquire the other

---

<sup>32</sup> Turner, *From Counterculture to Cyberculture*, 112.

“trappings” associated with one. “[A]cademic computer science departments, certification programs, and professional associations” all existed by 1968 and continue to exist today. Rather, Ensmenger identifies three primary causes of the failure to professionalize. First, programmers faced opposition from their corporate managers, to whom “professionalism was a potentially dangerous double-edged sword,” since the “last thing that traditional managers wanted was to provide data processing personnel with additional occupational authority.” Second, professional institutions such as the ACM “failed to convince employers of their relevance to the needs of business,” with the result that programmers derived little benefit from joining them. Finally, “internal rivalries within the computing community,” primarily between the academically-oriented programmers of the ACM, which emphasized the importance of establishing programming as a “well-founded” discipline, and the business-oriented programmers of the Data Processing Management Association (DPMA), which emphasized the need for certification programs, undermined programmers’ ability to counteract the external opposition they faced, and they “lost the leverage necessary to push through any particular professionalization agenda.” Here, again, the situation of programmers invites comparison to that of the New Communalists. The method the New Communalists employed in their quest for autonomy and for social transformation involved the use of tools to build alternative communities. Likewise, computer scientists aimed to use modular programming techniques as the practical and theoretical basis for the formation of a community of programming professionals, motivated not by the desire to enable managers to more easily control programmers, but by the desire to secure programmers’ autonomy and to mitigate the potential dangers of poorly-written software. Yet the community they created was too divided and, consequently, too weak to compel either society or the state to recognize its calls for professional status. This was not the last time programmers would attempt

to use modular programming as a tool to build a community and to assert their autonomy, though. Over the next two decades, the programming community would continue to be divided along various lines. However, modularity would gain near-universal acceptance among programmers as a mark of a well-written program and as a tool for sharing the products of their creativity. The ultimate expression of modularity's potential to enable such collaboration would come in the form of a community of programmers that, like the New Communalists before them, sought to assert their own autonomy and to transform society at large through the use of technology: the free software movement.<sup>33</sup>

---

<sup>33</sup> Ensmenger, *The Computer Boys Take Over*, 191, 193.

## **Embracing Modularity, Facilitating Creativity, and Challenging Property**

Over the course of the 1970s, a number of historic developments with links to both the New Communalists and the wider counterculture unfolded within the computing industry, from the emergence of “personal” computing to the creation of the first widely-accessible computer network communication systems. Among the more prominent of these developments was the rapid spread of the techniques and ideas associated with modular programming. By the early 1980s, nearly every programmer regarded modularity itself as a worthwhile goal, even if disagreements remained regarding the efficacy of particular techniques intended to enable or enforce modularity. Moreover, these programmers had come to link a program’s modularity with its aesthetic quality, suggesting that they saw modularity as a means by which programmers could express themselves to each other creatively. Their ideas regarding modular programming reflected its history as a methodology both formulated against a countercultural backdrop and propagated within a computing industry characterized by numerous connections to the lingering remnants of the counterculture. This history was again reflected in many programmers’ reactions when changes in the state of intellectual property laws allowed pieces of software to be copyrighted or patented, increasingly limiting programmers’ freedom to use modularity to express their creativity cooperatively and outside the control of any employer. Together, these three factors—programmers’ acceptance of modularity, their conception of it as a way to improve the aesthetic quality of their code and to express themselves creatively to other programmers, and their reactions to the new developments in intellectual property laws—set the stage for a new way of developing software through which programmers could be creative autonomously and cooperatively, making use of modularity and challenging the very idea of intellectual property in software.

## Attitudes toward Modularity

By the 1980s, programmers had already come to overwhelmingly embrace modularity as an idea, even to the point of regarding non-modular approaches to software development as in some sense backward. This shift occurred in the context of a 1970s computing industry many of whose key innovations were driven by individuals with connections to the 1960s counterculture. In 1973, the Xerox Palo Alto Research Center's Alto computer pioneered the mouse-driven graphical user interface that would come to be associated with "personal" computers. Among those responsible for its design was Alan Kay, the same computer scientist who coined the term "object-oriented" and regarded the *Whole Earth Catalog's* approach of providing tools through the use of which individuals could transform themselves and their communities as "the right idea." Slightly over two years later, Bob Albrecht, an early proponent of the idea that individuals could use computers as "tools [...] to enhance their own learning," published the specification for Tiny BASIC, a dialect of the BASIC programming language that would see widespread use in programs for the first generation of home computers, in his newsletter, the *People's Computer Company*. Albrecht would note in a 2001 interview that this newsletter, whose influential spin-off *Dr. Dobb's Journal of Software Tools* had published the founding document of the free software movement, Richard Stallman's "The GNU Manifesto," "was heavily influenced by the *Whole Earth Catalog*. I wanted to give away ideas." Meanwhile, the efforts of Lee Felsenstein, a computer engineer and participant in the Free Speech Movement at the University of California, Berkeley, anticipated the function of the Internet as a means of enabling peer-to-peer exchange of information, though not the precise technology behind it, by establishing a set of public terminals connected to a single time-sharing, multi-user computer in the San Francisco Bay area. This idea achieved its first truly large-scale implementation in 1980 with the creation of Usenet,

an early computer network communications system that might be informally described as a hybrid of email and of modern Internet forums. Unlike its close relative, the ARPANET, Usenet was accessible to anyone with the money to afford the service and the technical ability to use it. Together with the advent of personal computers, the creation of Usenet provided an opportunity for programmers to express their thoughts on a number of subjects, including modularity, to each other.<sup>34</sup>

During the early to mid 1980s, the net.lang newsgroup, a discussion group within Usenet that was focused on programming languages, was riddled with posts evaluating languages (and methods of implementing languages) by their tendencies to encourage or discourage modular programming. For example, one poster harshly criticized the facilities for modular programming provided by the C programming language, saying, while “[s]ome try to argue that ‘C’ is modular, simply because functions and data can be separately compiled,” the language “does not enforce any sort of binding within such modules” and, consequently, fails to effectively ensure “loose coupling” between modules.<sup>35</sup> Another poster, while defending the relative merits of interpreted languages, whose programs are read and executed directly by a program called an “interpreter,” when compared with compiled languages, whose programs are first translated into machine-executable code by a program called a “compiler,” did so on the grounds that “techniques of modularity, information hiding, levels of abstraction and simplicity can be applied with an interpreted language as well, if not better, than a compiled one.”<sup>36</sup> A third poster, on a related note, asked, “Do interpreters teach bad habits?” and concluded that the problem was not “the interpreter as much as the language. BASIC, for example, [...] makes writing modular code

---

<sup>34</sup> Turner, *From Counterculture to Cyberculture*, 112-115.

<sup>35</sup> Jan Steinman [jans@mako.UUCP], “RE:Definition of Buzzwords: ‘Object-Oriented,’” in [net.lang], 23 January 1985, <http://groups.google.com/group/net.lang/topics>.

<sup>36</sup> Dave Newkirk [dcn@ixn5h.UUCP], “Re:Interpreters vs. Compilers – Who wins?” in [net.lang], 23 March 1983, <http://groups.google.com/group/net.lang/topics>.

difficult. [...] My friends who were weaned on BASIC tend to produce sloppier code than those who were weaned on a structured language. That may just be their own fault, but I think the language has something to do with it.”<sup>37</sup> As this comment suggests, programmers did not limit themselves to judging languages based on their support for modular programming: they also evaluated other programmers and themselves based on their tendency to program in a modular manner. For instance, while describing a data structures course he had taken one summer at the University of Wisconsin, another poster credited it with having “broke[n] me of my worst habits,” which he listed as “poor modularity, too few comments, variables called ‘x,’” and, with a touch of irony, “a good night’s sleep.”<sup>38</sup> This recalls the original purpose motivating the computer scientists who articulated the importance of modularity in programming during the 1960s: to provide standards by which the community of programmers could self-regulate and, thus, professionalize.

Programmers likewise exhibited a near-universal acceptance of some particular modular programming techniques, such as structured programming. Consider, for instance, programmers’ attitudes toward unstructured “spaghetti code,” or code with a complex flow of execution resulting from the “go to” statements that Edsger Dijkstra had condemned in the late 1960s. One Usenet poster complained about how the programming language FORTRAN’s characteristics encouraged such “spaghetti code,” saying that “I have grown tremendously intolerant of FORTRAN in the last few years for its come-hither-and-write-spaghetti attitude. [...] If you live in the shadow of decades of FORTRAN, heaven help you and pass the Parmesan.”<sup>39</sup>

---

<sup>37</sup> Alan Hu [ajh@sdcsvax.UUCP], “RE:Difficulty of recursion,” in [net.lang], 25 March 1983, <http://groups.google.com/group/net.lang/topics>.

<sup>38</sup> Ken Perlow [ken@ihuxq.UUCP], “RE:Professionalizing Programmers, Going Off-line,” in [net.lang], 8 June 1984, <http://groups.google.com/group/net.lang/topics>.

<sup>39</sup> George Sherouse [sherouse@unc.UUCP], “RE:Unix for physicists (attn:finn),” in [net.physics], 14 June 1984, <http://groups.google.com/group/net.physics/topics>.

Interestingly, this statement implied more than just impatience with “spaghetti code.” FORTRAN is one of the oldest compiled programming languages, having been originally developed in 1957, and as of 1984, it lacked many of the constructs newer languages used to prevent programmers from writing “spaghetti code” and to encourage and enable a more modular coding style. By identifying spaghetti code as characteristic of FORTRAN programs, this statement implied that the poster viewed such code as a relic of an earlier period in software development, before modular programming techniques came to be widely-used by programmers and widely-supported by programming languages. Another poster boldly declared that “I think every line of a C or assembler program should be documented even if you as a programmer think the code is so obvious anybody could understand it. I came to this conclusion after years of trying to decipher other people's spaghetti.”<sup>40</sup> The idea that every section of code should be documented with an explanation of its purpose itself implied adherence to the modular paradigm. In the absence of such adherence, the divisions of the code into discrete blocks with well-defined purposes would be impossible. All that said, some programmers did defend the use of “go to” statements in certain circumstances. One noted that “In a language like Modula-2 where goto is not supported, there is no way to code the exceptional handling procedures nicely as there is no exceptional handling construct either,” or, in other words, there is no way for execution to quickly jump to a block of code designed for handling errors when an error occurs.<sup>41</sup> However, even this example did not represent a rejection of structured programming. Rather, it represented an acknowledgement that, within a program that generally uses control structures to avoid the complexities associated with “go to” statements, “go to” statements might be still be necessary if

---

<sup>40</sup> John Crane [crane@fortune.UUCP], “RE:Self-modifying code,” in [net.lang], 3 January 1984, <http://groups.google.com/group/net.lang/topics>.

<sup>41</sup> C.J. Lo [cjl@iuvox.UUCP], “Gotos, tail-recursion,” in [net.lang], 20 June 1984, <http://groups.google.com/group/net.lang/topics>.



the provided control structures prove insufficient—because, for example, of the lack of an “exceptional handling construct.”

Other modular programming techniques lacked the level of acceptance that structured programming enjoyed. However, just as the above defense of “go to” statements did not constitute a rejection of structured programming, so did these critiques of various modular programming techniques not constitute rejections of modular programming in general. In particular, one poster, criticizing the performance penalty that programs written using the technique known as “object-oriented programming” often took, countered the claim that “programming languages are designed to express ideas” (a claim that he implicitly associated with advocates of object-oriented programming) with the claim that “programming languages [...] are ways of telling the computer WHAT TO DO” and went on to note that “object-oriented programming is only ONE model of abstraction, and isn't the most accepted one. It would be accepted more if there were implementations that performed as well as the others.”<sup>42</sup> Yet this criticism in itself implied support for “abstraction” in general, as long as that abstraction did not come about through object-oriented programming. This support for abstraction, meanwhile, implied support for modularity in general, since the underlying implementation of an abstraction would necessarily constitute a module—an independent section of code, decoupled from the rest of the program. In rejecting object-oriented programming, then, this programmer affirmed his acceptance of modular programming.

Much as the *Whole Earth Catalog* had “served as a textual forum within which back-to-the-landers could meet one another, as well as technologists, academics, and artists, and share information,” Usenet served as a forum through which programmers could exchange information

---

<sup>42</sup> Dave Brownell [brownell@harvard.ARPA], “RE:Object oriented (flames at end),” in [net.lang], 21 June 1984, <http://groups.google.com/group/net.lang/topics>.

and ideas with each other.<sup>43</sup> As this exchange makes clear, the programmers posting on Usenet had come to accept modularity as a worthy goal, and even those posters who rejected some particular method of achieving modularity embraced modularity itself. Their reasons for doing so reflected modular programming's history as a technology conceived in the midst of the 1960s and propagated within a computing industry heavily influenced by the counterculture.

### **Modularity and the Art of Programming**

That programmers had come to embrace modular programming by the early 1980s says little about the purposes to which they sought to put modular programming. A more careful examination of the written material produced by these programmers indicates that they saw modularity as a way of not only simplifying the task of maintaining large software projects, but also as a tool to use to create aesthetically-pleasing code in much the same way that countercultural artists such as USCO saw “multimedia slide, light, and sound shows” as tools to convey sensations of unity by purportedly “plug[ging] in to mystical currents that flowed among the group members and within each of them.”<sup>44</sup> This conceptualization of programs as works of art with aesthetic qualities had precedents in the material produced by the computer scientists of the late 1960s and early 1970s. The computer scientist Donald Knuth, known for, among other contributions, creating the TeX typesetting system and for introducing and popularizing formal mathematical techniques for analyzing the efficiency of algorithms, began writing an influential series of books called *The Art of Computer Programming* in the 1960s. Upon receiving the ACM's Turing Award in 1974, Knuth explained the title of his series by saying that “[w]hen I speak about computer programming as an art, I am thinking primarily of it as art *form*, in an aesthetic sense,” that “[t]he chief goal of my work as educator and author is to help people learn

---

<sup>43</sup> Turner, *From Counterculture to Cyberculture*, 79.

<sup>44</sup> *Ibid.*, 50.

how to write *beautiful programs*,” and, referencing the thoughts of Andrei Ershov, that much like “composing poetry or music,” “programming can give us both intellectual and emotional satisfaction, because it is a real achievement to master complexity and establish a system of consistent rules.” Knuth made haste to note that “there is no one ‘best’ style” in programming and that “the important thing is that programmers are creating something *they* feel as beautiful.” However, citing utilitarian philosopher Jeremy Bentham, he also noted the existence of “certain principles of aesthetics which are better than others, namely the ‘utility’ of the result.” Although “[w]e have some freedom in setting up our personal standards of beauty” and, indeed, the pleasure derived from creating beautiful programs represented an “incontestable” form of “utility” in itself, “it is especially nice when the things we regard as beautiful are also regarded by others as useful.” In particular, he identified a program as “useful” to others when it “works correctly” and “won’t be hard to change, when the time for adaptation arrives,” both of which “are achieved when the program is easily readable and understandable to a person who knows the appropriate language.” Furthermore, a key part of his conception of programming as an art form was the fact that “when we read other people’s programs, we can recognize some of them as genuine works of art,” which in turn would be dependent on these programs being readable. It was precisely these aesthetic considerations—the clarity or readability of code—that animated the material written by programmers on the subject of modular programming during the early 1980s.<sup>45</sup>

In the early 1980s, Usenet featured a number of widely-circulated, tongue-in-cheek posts about “real programmers,” programmers from an earlier period who had not yet adopted common practices such as the use of modular programming techniques or even, in some cases,

---

<sup>45</sup> Donald Knuth, “Computer Programming as an Art,” *Communications of the ACM* 17, no. 12 (1974): 670-671, doi:10.1145/361604.361612.

high-level programming languages, languages that simplify programming by abstracting away the details of a computer's instruction set. One of these posts declared, with regard to programs written by "real programmers," that "if you throw them on the machine they can be patched into working in 'only a few' 30 hour debugging sessions," an ironic statement indicating the belief that the non-modular code that might be written by a "real programmer" would be difficult for other programmers to read and maintain.<sup>46</sup> The popularity of such posts indicates a widespread belief that modularity made code more readable, which in turn made life more convenient for programmers who had to work with the code of other programmers. This would have constituted a fairly obvious reason for programmers, following Donald Knuth's example in considering "useful" standards of beauty to be "better" than alternative standards, to embrace readability as an aesthetic criterion and modular programming as a means of fulfilling that criterion.

One of these posts in particular indicates something less obvious and more interesting about the ways in which at least some programmers believed modular programming could be applied: that by making code more readable, modularity would make the creativity that went into writing the code easier for a reader to perceive. The post in question (now a fairly famous piece of hacker folklore known as "The Story of Mel") related the tale of the poster's attempt to modify a machine-language program written by a "real programmer" named Mel. Ultimately, the poster "quit looking" because "I didn't feel comfortable hacking up the code of a Real Programmer."<sup>47</sup> This phrase had a double meaning: on the one hand, the poster felt that "hacking up" the code would be a violation of the original coder's artistic integrity, and on the other hand, the poster simply found the process too exhausting to endure. By linking the latter meaning with

---

<sup>46</sup> Jim Livingston [jiml@pesnta.UUCP], "RE:REAL PROGRAMMERS," in [net.jokes], 7 September 1984, <http://groups.google.com/group/net.jokes/topics>.

<sup>47</sup> Matt Crawford [matt@oddjob.UChicago.UUCP], "RE:The realest programmer of all," in [net.jokes], 20 November 1984, <http://groups.google.com/group/net.jokes/topics>.

the former and declaring, “I have often felt that programming is an art form” featuring “lovely gems and brilliant coups hidden from human view and admiration, sometimes forever, by the very nature of the process,”<sup>48</sup> the post implied that writing code in a non-modular fashion served to obscure the creativity that went into the writing of that code. In a sense, the ingenuity and artistry of “real programmers” who wrote non-modular code was wasted, lost to history because of other programmers’ inability to read the code in question. Modularity, then, might serve as a means by which programmers could make their art more apparent and more accessible to others—a notion that one poster expressed by saying that a “programming language may be considered just a means of communicating with a computer, but it should not be forgotten that it is also a means of communicating with other programmers (including the original author, say in three weeks or so).”<sup>49</sup> This is consistent with the important role Knuth assigned to the act of recognizing others’ programs as “genuine works of art” in his argument for regarding programming as an art. It also sheds some light on why Andrei Ershov regarded “the highest aesthetic idea of our profession” to be “to make the art of programming public property, and thereby to submerge our elite exclusiveness within a mature mankind,” as the extent to which the creativity of any given program would be appreciated and acknowledged would be proportional to the number of people capable of comprehending it and, consequently, to the extent to which programming was “public property.”<sup>50</sup>

Most Usenet posters seemed to have followed the approach of the previously-mentioned posters in regarding modularity as making programs aesthetically pleasing as a result of making them readable. However, at least one poster went beyond this claim, instead directly equating the

---

<sup>48</sup> *Ibid.*

<sup>49</sup> Dan Frank [g-frank@gumby.UUCP], “RE:High-levelity,” in [net.lang], 26 December 1984, <http://groups.google.com/group/net.lang/topics>.

<sup>50</sup> Ershov, “Aesthetics and the Human Factor in Programming,” 505.

modularity of a program with aesthetic qualities such as beauty, elegance, or expressivity. Arguing against another's proposal that a windowing system be included as part of the core functionality of the Unix operating system (i.e., the Unix "kernel") while sarcastically identifying as "another programming-as-art nut," claimed that the "beauty" of Unix "is that the kernel contains the minimum amount of code to do what a good operating system should do" and that a feature of Unix the other programmer had identified as a problem, "the lack of knowledge that the kernel has about the [user's] environment [...] simply is not a problem, but rather a benefit," since it meant that "the kernel does not have to be tailored to meet specific requirements."<sup>51</sup> The beauty of Unix, in other words, lay precisely in the modularity of its design, with all but its core functionality being kept out of the main program itself and implemented as separate applications.

Such sentiments appeared not only in programmers' praise of programs they believed to be clear and elegant, but also in their castigation of programs they believed to be unreadable and inelegant. After one programmer proposed a contest which would present an award to the person capable of writing the "Most Disgusting Code," others quickly rejected the notion on the grounds that, first, "good code (defined as portable, readable and maintainable) being much rarer, much more valuable to look at (and learn from), and (unfortunately) alot harder to find," "perhaps what we need is a 'Best Code I've ever seen' award," and, second, such a contest actually already existed, and was known as the "International Obfuscated C Code Contest."<sup>52</sup> The name of the existing contest is particularly significant. It equates "disgusting" code with "obfuscated" code, implying that good code would be defined by its readability and, as a result, by its modularity.

---

<sup>51</sup> P.D. Guthrie [pdg@ihdev.UUCP], "RE:Windowing systems," in [net.unix-wizards], 16 April 1986, <http://groups.google.com/group/net.unix-wizards/topics>.

<sup>52</sup> Joe Boykin [boykin@datagen.UUCP], "RE:Disgusting Code," in [net.unix-wizards], 9 February 1986, <http://groups.google.com/group/net.unix-wizards/topics>.

On a related note, another programmer reflected that “many so-called coders have little or no circumspection or (to rob from D. Knuth) art to what they are doing,” and that “to be genuinely creative requires background and a developed ability to intuit an elegant solution. Else all you get is pretty shitty code cranked out by somebody who doesn’t care and doesn’t appreciate the overall quality and elegance (there’s that word again!) of the program or system that is developed.”<sup>53</sup> Here, again, the notion that “good” code is elegant, creative, and artistic is the key, underlying sentiment.

This raises the question: why would the aesthetic features of code matter so much to programmers? One programmer, reflecting “all this talk of ‘good’ and ‘bad’ code,” proposed that “we programmers,” in contrast to “managers who more often than not emphasize short-term goals” and “probably don’t care how the code ‘looks’ as long as it ‘works,’” “have to ‘look’ at the code, often for long hours seemingly without end,” with the result that it “is on [an] aesthetic level that code is often judged fish or fowl” (though he goes on to say that “one may argue that code really doesn’t ‘work’ when it ‘looks’ bad. This often comes into play when someone, usually not the original author, must ‘look’ at such code, and ‘fix’ it”).<sup>54</sup> For him, then, the desire to make code aesthetically-pleasing was simply the result of programmers’ desire to make their jobs more enjoyable by making code more pleasant to read.

Yet this was not the only implication of programmers’ attempts to use modularity to craft aesthetically-pleasing code and of their conceptualization of clear and readable code as aesthetically-pleasing. By implying that programmers ought to view programming as pleasurable in itself and, in Knuth’s words, implying that they “shouldn’t shy away from ‘art for art’s sake’”

---

<sup>53</sup> Mark Kenig [cbspt002@abnjh.UUCP], “RE:Teaching programming –GOTO’s, Object Oriented Languages, etc.” in [net.lang], 20 June 1984, <http://groups.google.com/group/net.lang/topics>.

<sup>54</sup> Glenn Adams [glenn@LL-XN.ARPA], “RE:code quality,” in [net.unix-wizards], 9 October 1985, <http://groups.google.com/group/net.unix-wizards/topics>.

and “shouldn’t feel guilty about programs that are just for fun,” they implied that programmers ought to see programming, particularly when undertaken autonomously, as a rewarding act of creativity not unlike crafting music or poetry.<sup>55</sup> By implying that programmers ought to strive for an aesthetic ideal that emphasized the importance of their code’s comprehensibility to other programmers, they implied that programmers might be able to collaborate effectively on creative endeavors involving the construction of large amounts of code. Finally, by stressing the importance of programs’ readability and adaptability, they implied that programmers ought to be free to read and adapt programs. The conceptualization of programming as having aesthetic qualities identified as readability and clarity, in other words, lent support to the values that would soon come to be associated with the free software movement and the growth of a programming subculture that would be willing to collectively undertake large programming projects with little incentive other than the love of programming. However, the first such projects did not occur under conditions lacking incentives other than the love of programming. Rather, they occurred in the context of what many, though by no means all, programmers perceived as a threat to the core values of programming as an art.

### **Reactions to Proprietary Restrictions on Software**

During the 1980s, barriers resulting from the application of intellectual property law to the field of software development began to impede the previously-unrestricted redistribution of code, limiting the one of the chief benefits modular programming techniques were supposed to bestow on programmers—the ability to create new code using preexisting software as a base. As a result of these new barriers, such creation could often only occur when the programmer held a license to use the preexisting software in question, usually when he or she took on the role of employee to an employer able and willing to purchase such a license. Consequently,

---

<sup>55</sup> Knuth, “Computer Programming as an Art,” 672.



programmers faced an environment in which their ability to create software outside the control of managers grew ever more restricted. They reacted to these restrictions in a sharply polarized manner that echoed a controversy from the 1960s: the debate over the permissibility of classified research in a university environment.

According to Stuart Graham and David Mowery, the increasing importance of “packaged” software (often, though not always, distributed as proprietary modules that, once licensed, could be used in other programming projects, in a manner reminiscent of the “software component libraries” Douglas McIlroy envisioned), together with the increasing degree to which software was decoupled from hardware, drove the software industry’s increasing reliance on intellectual property protection. During the early years of the industry, they contend, pieces of software were generally tied to particular pieces of hardware. The lack of standardized computer architectures or of portable high-level programming languages prevented software from being developed and run on many different machines, with the result that “most of the software for early mainframe computers was produced by their manufacturers and users.” As these conditions changed, “independent software vendors” emerged that produced software that could run on many different computers even as hardware producers “unbundled” their software products from their hardware products, distributing each separately. The advent of large-scale computer networks and, eventually, the Internet accelerated this process, lowering barriers to entry for independent software vendors by providing a low-cost means of distributing software that was not controlled by the hardware producers against whom they competed.<sup>56</sup>

As “packaged” software became increasingly important to the profits of both independent software vendors and companies that had originally been hardware producers, securing and

---

<sup>56</sup> Stuart Graham and David Mowery, “Intellectual Property Protection in the U.S. Software Industry,” in *Patents in the Knowledge-Based Economy*, Washington, DC: The National Academies Press (2003), 220-222.

restricting the rights to distribute this software likewise became increasingly important. To this end, vendors initially relied on their products' status as trade secrets, which they attempted to defend by requiring their employees and the users of their software to sign non-disclosure agreements. This approach granted them a legal basis for protecting their software from industrial espionage and from public redistribution by its users and developers. However, if the software ceased to be secret through legitimate means—including independent invention, reverse-engineering, and the lack of sufficient efforts to maintain its secrecy on the part of the vendors—this legal protection would disappear. In the *Copyright Act* of 1976, the United States Congress had, among other provisions, granted copyright holders the exclusive right to reproduce, distribute, and create derivative works from copyrighted works either for the duration of the creator's lifetime plus fifty years or, in the case of works made for hire, for exactly seventy-five years. Perhaps with the limitations of trade secret protection in mind, policymakers "singled out" copyright protection "as the preferred means for protecting software-related intellectual property," leading the United States Congress to amend the *Copyright Act* in 1980 to include a definition of "computer program." Software copyright soon saw application at the federal judicial level in the 1983 case *Apple Computer, Inc. v. Franklin Computer Corp.*, in which the court held that the operating system for the Apple II computer was protected by copyright even if compatibility with existing software could only be achieved by copying it, as well as in later decisions that extended the protection that copyrighted status conferred on a piece of software from the "traditional copyright protection of *expression* to such non-literal elements of software as structure, sequence, and organization." The scope of patent law was expanded to include certain aspects of computer programs around the same time. In the 1981 cases *Diamond v. Diehr* and *Diamond v. Bradley*, the United States Supreme Court ruled that "inventive

concepts” or algorithms underlying a piece of software could be patented—a marked departure from 1972’s *Gottschalk v. Benson*, which suggested the opposite. The 1990s would see some of the extensions of the meaning of copyright protection in the case of software overturned. For instance, in *Lotus Development Corporation v. Borland International, Inc.*, the Supreme Court affirmed a lower court’s decision that the imitation of the “look and feel” of a copyrighted program for the sake of compatibility did not constitute infringement. However, court decisions “consistently broadened and strengthened the economic value of software patents.”<sup>57</sup>

Many programmers reacted to these developments with a great deal of skepticism. As one programmer argued, “when a company takes and declares restricted under trade secrets things which are just not reasonable to so do, it is more than reasonable to argue with them,” objecting in particular to the idea that someone could “suddenly” declare “top-down parsing” to be a “trade secret because it is used in some part of Unix (must be somewhere)” and holding that “creative works should be protected but not using that route.”<sup>58</sup> This programmer was not objecting to intellectual property in software per se. Rather, he was objecting to the idea that what many programmers would consider a fundamental technique of computer science could be owned, to the notion that the use of an algorithm could be restricted by law. On a similar note, others expressed disbelief at Bell Labs’ claiming copyright protection of the code for extremely simple Unix programs. For example, in response to one programmer’s post containing implementations of `\bin>true` and `\bin>false`, which respectively do nothing “successfully” and do nothing “unsuccessfully,” another programmer declared, “A clear case of independent creation! An idea whose time has come! (Of course, the question can be asked, ‘What other possible reasonable way could there be to do it?’),” ridiculing Bell Labs’ ownership claim over the code in

---

<sup>57</sup> *Ibid.*, 223-225.

<sup>58</sup> Dave Farber [farber@udel-huey.ARPA], “RE:software ethics,” in [net.unix-wizards], 3 March 1985, <http://groups.google.com/group/net.unix-wizards/topics>.

question.<sup>59</sup> Perhaps the simplest expression of these related objections came from the same thread, when one poster stated, “You cannot copyright ideas.”<sup>60</sup> (This statement was literally true, as only expressions of ideas are eligible for copyright. However, as another poster responded, equally-truthfully: “Sure, but you can use them [ideas] as the basis as a patent, can you not?”)<sup>61</sup>

These objections in particular have much in common with the objections many university faculty members raised regarding their universities’ acceptance of classified research contracts from government agencies—often, the Department of Defense—during the period from World War II to the Vietnam War. The results of research funded by these contracts were kept secret, and university personnel participating in these projects had to be investigated by the Department of Justice to determine whether or not they posed a “security risk.”<sup>62</sup> Though some faculty supported this secrecy on the grounds that it contributed to national security, many regarded it as fundamentally compromising the goal of the “modern research university where free and open exchange of ideas is the hallmark of higher learning,” and, by the end of the war, “there was a broad consensus among major research universities that classified or secret research was incompatible with the values of academic science,” since “the pursuit of certifiable knowledge requires transparency to actualize the self-correcting function of science.”<sup>63</sup> Much as these programmers saw the potential for new developments in IP law to conflict with their basic values, such as the notion that algorithms and programs regarded as “fundamental” ought to be

---

<sup>59</sup> Eric Black [eric@chronon.UUCP], “RE:lex and yacc in the public domain (responses),” in [net.unix-wizards], 1 May 1986, <http://groups.google.com/group/net.unix-wizards/topics>.

<sup>60</sup> David Gast [gast@ucla-cs.ARPA], “RE:lex and yacc in public domain (responses),” in [net.unix-wizards], 3 May 1986, <http://groups.google.com/group/net.unix-wizards/topics>.

<sup>61</sup> Gregory Smith [greg@utcsri.UUCP], “RE:lex and yacc in public domain (responses),” in [net.unix-wizards], 6 May 1986, <http://groups.google.com/gropu/net.unix-wizards/topics>.

<sup>62</sup> Sheldon Krinsky, “When Sponsored Research Fails the Admissions Test,” in *Universities at Risk: How Politics, Special Interests, and Corporatization Threaten Academic Integrity*, ed. James Turk (J. Lorimer & Co., 2008), 80.

<sup>63</sup> *Ibid.*, 80-81.

freely available for all programmers to use, so did the academics objecting to classified research see restrictions on the publication of research results as conflicting with the basic values of modern, open universities. There was even at least one concrete connection between the two groups. Richard Stallman, the future founder of the Free Software Foundation, an organization dedicated to the advancement of non-proprietary software, worked as a programmer at the Massachusetts Institute of Technology's Artificial Intelligence Lab during and after his time as an undergraduate at Harvard University in the 1970s. As a first-year student, he had been known for his strong performance in Harvard's infamously-difficult class Math 55, but after graduating, he became known for his activism. When the Defense Department attempted to initiate the use of passwords on the system in the lab to restrict access to the information on the ARPANET, Stallman and a number of other programmers in the lab changed their passwords to the empty string of characters as part of Stallman's "guerilla war" against restrictions on the sharing of information.<sup>64</sup>

Not all programmers who objected to the application of intellectual property restrictions to software did so on the grounds of their implications for the values associated with programming. On examining a set of Unix source and documentation files and finding that few had been marked as copyrighted despite the mixed copyright status of these files, one programmer asked, "How are we poor innocent programmers to know what's copyrighted and what isn't if BTL [Bell Telephone Laboratories, the producers of Unix] doesn't bother marking the stuff and there's non-copyrighted stuff in the same directories? I'm perfectly willing to respect everyone's copyright (even the 'big bad guys' like IBM and BTL), but I think I'm entitled

---

<sup>64</sup> Roy Rosenzweig, "Wizards, Bureaucrats, Warriors, and Hackers: Writing the History of the Internet," *The American Historical Review* 103, no. 5 (1998): 1542.

to fair notice!”<sup>65</sup> He then elaborated on the reasons such conditions were problematic: “Awhile back a fellow netter called me to task for including the entire text of dd.c along with a fix I was sending out to net.sources. I looked back at it. NOT ONE WORD ABOUT COPYRIGHT appears anywhere in the original source.” His objection, then, focused more on the difficulties that introducing intellectual property to a software industry in which code had traditionally been freely redistributable imposed on programmers. He was not alone in having such concerns, either. Another programmer expressed his concerns about the implications of the proprietary status of Unix tools, asking whether it was true that “the object [machine] code from a proprietary C compiler is itself proprietary” and that “any program written in yacc is proprietary, because the algorithms output by yacc are proprietary.”<sup>66</sup> Together, these would imply that he would be unable to distribute software produced using these two programs “without requiring that the site I distribute the software to has a Unix license,” an idea that he had “trouble adjusting to [...] coming from a world (DEC-20) where no matter how proprietary the compiler may be, the ownership of the executable binaries belongs to the owner of the source code even if some library routine from the compiler’s runtime is used.”<sup>67</sup>

That said, programmer reaction to these developments in intellectual property law was far from unanimous. One programmer, for example, responded to the first of the above objections (that is, the assertion that software intellectual property protections might problematically allow companies to claim ownership of fundamental algorithms or of obvious approaches) by saying, “If someone came up with a new way of doing top-down parsing that took substantial work on their part and represented a potential commercial advantage, I see nothing wrong with their

---

<sup>65</sup> Barry Gold [barryg@sdcrcf.UUCP], “RE:Where’s the (c) on unix?” in [net.unix-wizards], 21 March 1984, <http://groups.google.com/group/net.unix-wizards/topics>.

<sup>66</sup> Mark Crispin [Admin.MRC@SU-SCORE.ARPA], “RE:public domain?” in [net.unix-wizards], 26 December 1984, <http://groups.google.com/group/net.unix-wizards/topics>.

<sup>67</sup> *Ibid.*

placing it under trade-secret protections.”<sup>68</sup> Earlier, she had claimed that “It's attitudes like yours that have forced vendors into ever more restrictive agreements, copy protection schemes, and other similar sorts of things to try protect themselves” and that “I treat other people's software and trade secrets the same way I expect them to treat mine, which is to say I respect them.”<sup>69</sup> Another programmer, responding to her, equated intellectual property restrictions on the distribution of software with ensuring that programmers were compensated for their labor, saying, “If, as our economic system and culture embrace, people are to benefit from their labors [...] then one must include programmers' labors, too.”<sup>70</sup> Finally, a third programmer responded to a fourth's inquiries after public domain versions of the proprietary Unix programs lex and yacc with “I am amazed. You \*know\* that these things are somebody else's property. You \*know\* that they don't want you to take them. Yet people can quibble over whether there were proper copyright notices or if trade secrets were properly enforced,” asking, “If you see somebody's money on the ground, do you take it if you can get away with it? If it isn't tagged but you know who it belongs to, do you take it?”<sup>71</sup>

Much as the introduction of classified research to university campuses failed to meet uniform opposition among university faculty during the late 1960s, the imposition of increasingly restrictive constraints on the distribution of software failed to meet uniform opposition among programmers during the 1980s. On the contrary, many programmers considered the use of patent, copyright, or trade secret protections to control the distribution of code to be as justifiable as the ownership of a tangible good. Moreover, those that did oppose

---

<sup>68</sup> Lauren Weinstein [lauren@rand-unix.ARPA], “RE:software ethics,” in [net.unix-wizards], 3 March 1985, <http://groups.google.com/groups/net.unix-wizards/topics>.

<sup>69</sup> Lauren Weinstein [lauren@rand-unix.ARPA], “RE:software ethics,” in [net.unix-wizards], 2 March 1985, <http://groups.google.com/groups/net.unix-wizards/topics>.

<sup>70</sup> Ed Gould [ed@mtxinu.UUCP], “RE:software ethics,” in [net.unix-wizards], 21 March 1985, <http://groups.google.com/groups/net.unix-wizards/topics>.

<sup>71</sup> Brad Templeton [brad@looking.UUCP], “RE:lex and yacc in the public domain (responses),” in [net.unix-wizards], 7 April 1986, <http://groups.google.com/groups/net.unix-wizards/topics>.

these constraints were much less successful than the university professors that opposed classified research. Though major universities reached a consensus that classified research was incompatible with the values of academic science, no corresponding consensus emerged among programmers with regard to intellectual property in software, and the use of both copyrights and patents remained common in the software development industry.<sup>72</sup> Nevertheless, the opponents of intellectual property in software remained vocal and passionate in their opposition, and, whether it was rooted in a belief that the ownership of the ideas underlying a program was inherently unjustifiable or simply in a pragmatic evaluation of the impact such constraints would have on their ability to continue to reap the benefits of modular, reusable code as they had previously, this opposition ultimately represented resistance against policies that promised to limit programmers' autonomy. These beliefs, coupled with the impulse to create, would soon motivate the many programmers who would come to contribute to the emergence of an alternative model of software development, one which would challenge the idea of intellectual property in software and which modularity would make possible.

\* \* \* \* \*

The computer scientists of the late 1960s had introduced the ideas and techniques associated with modular programming in hopes that doing so would allow programmers to professionalize and to secure their autonomy. When the programmers of the 1970s and early 1980s adopted these techniques, they did so largely because they believed they had the potential to make their code clearer, more readable, and, thus, both more useful to other programmers and more pleasing to behold. Growing to regard readability and clarity as key criteria by which to judge the aesthetic quality of code, they followed Donald Knuth in identifying aesthetic criteria with a tendency to produce useful results as “good” aesthetic criteria. In the process, many—

---

<sup>72</sup> Krimsky, “When Sponsored Research Fails the Admissions Test,” 80-81.



reflecting the countercultural influences on the development of modular programming and other developments in the computing industry—came to regard programming as an art form, a creative act that was rewarding in itself. For those that did, the qualities by which they judged programs aesthetically, readability and clarity, encouraged them to view large-scale collaboration among programmers as both possible and worthwhile and to believe that programmers ought to be free both to share their programs with others and to adapt programs written by others. These values were not without precedent in the computing community, as they shared much in common with the “hacker ethic” that the journalist Steven Levy identified as having originated among the programmers at MIT’s Artificial Intelligence Laboratory during the 1950s and 1960s, which in turn closely paralleled the countercultural values of the New Communalists: that “access to computers [...] should be unlimited and total,” that “all information should be free,” that programmers should “mistrust authority” and “promote decentralization,” that “you can create art and beauty on a computer,” and that “computers can change your life for the better.”<sup>73</sup> Yet with the mass adoption of modularity as a tool by which to improve the aesthetic quality of code, these values were able gain traction far beyond the confines of MIT’s AI Lab, laying the foundation for a large, geographically-distributed community of adherents to emerge on network forums such as Usenet. When changes in the United States’ body of intellectual property law imposed new restrictions on programmers’ freedom to share and adapt programs, threatening the core values embedded in both the “hacker ethic” and the conceptualization of programming as an art form, many programmers who had internalized these values resisted. At first, this resistance took the form of Usenet posts arguing against the legitimacy or the wisdom of proprietary restrictions on the distribution of software. Soon, though, it would take the form of a series of

---

<sup>73</sup> Steven Levy, *Hackers: Heroes of the Computer Revolution*, (Garden City, New York: Anchor Press/Doubleday, 1984), 26-31.

attempts to put the “hacker ethic” into practice on a massive scale and in a manner that would directly undermine patent and copyright holders’ attempts to restrict the freedoms programmers had previously enjoyed. At the head of the first of these efforts, the GNU Project, was one of the “hackers” of MIT’s AI Lab: Richard Stallman.

## Free, Open Source, and Modular Software: The Rise of Cooperative Development

“Free software,” according to Richard Stallman’s definition, “is a matter of liberty, not price,” with the “free” in its name meaning “‘free’ as in ‘free speech,’ not as in ‘free beer,’” although his stringent definition of free software ensured that virtually all programs considered “free” in the first sense were also free in the second sense. When Stallman announced his intention to create a replacement for the Unix operating system in 1983, he also announced that the product of his efforts, GNU, would be “free software” as he formulated it. Such software was characterized by four specific freedoms: the freedoms “to run the program, for any purpose,” “to study how the program works, and adapt it to your needs,” “to redistribute copies so you can help your neighbor,” and “to improve the program, and release your improvements to the public, so that the whole community benefits.” These freedoms were integral to the “hacker ethic” of the community of programmers at MIT’s Artificial Intelligence Laboratory from which Stallman came. Moreover, they were precisely the freedoms that intellectual property law had begun to threaten during the 1980s and that Stallman created the GNU Project and the Free Software Foundation to protect. The 1980s and 1990s would see the emergence of large, geographically-distributed groups of programmers who would use modular programming as a tool to enable them to collaborate on complex, freely-distributable software projects such as GNU. In the process, these programmers would build a community that—despite internal divisions between those who favored the term “free” and those who favored the term “open source” to describe freely-distributable software—embraced a common vision of autonomy and cooperation and, like the countercultural alternative communities built by the New Communalists before it, believed in its potential to remake society in its own image.<sup>74</sup>

---

<sup>74</sup> Richard Stallman, “Free Software Definition,” in *Free Software, Free Society: Selected Essays of Richard M. Stallman*, ed. Joshua Gay (Boston: GNU Press, 2002), 43.

## **The Role of Modular Programming in a Distributed Development Model**

The idea behind GNU was straightforward: programmers interested in preserving Stallman's four freedoms would collaborate to create an operating system compatible with the ubiquitous, proprietary, and highly modular Unix operating system that would not be derived from any existing Unix code. The resulting product would then be "copylefted," that is, copyrighted and distributed under a license (the GNU General Public License) that would grant all four freedoms Stallman associated with free software while also imposing the requirement that any programs incorporating GNU code be distributed under the same license. GNU would thus provide a starting-point for the development of a wide variety of free software, beginning with essential programs such as operating systems and then proceeding to include less-essential applications as well. Eventually, free software might even approach the point that, for every proprietary program containing pieces of code a programmer might want to reuse or modify, a free alternative would exist, with the result that the freedom that programmers had experienced prior to the 1980s would be effectively restored. For this to succeed, a sufficiently large number of programmers would have to be willing to donate their time and effort to the creation of free software. More than that, though, the collaboration of such a large number of programmers, likely separated from each other by vast geographical distances, on complex software projects would have to be feasible in the first place. Here, there exists a parallel between the free software movement's use of the provisions of already-existing copyright law and of modular programming (both of which, if Philip Kraft's argument is to be believed, may have contributed to a reduction of programmer autonomy in the past): much as the General Public License took advantage of provisions of already-existing copyright law to ensure that software derived from GNU would remain free, so would programmers make use of modularity to enable themselves to

work independently on pieces of enormous, complicated programs, which, when completed, could be combined to form a freely-accessible whole.

The free software movement's reliance on modular programming techniques to make its distributed development model possible—and, for that matter, its advocates' acknowledgement of this reliance—stretches back to Stallman's founding of the GNU Project. In "The GNU Manifesto," his 1985 essay appealing to programmers for support and participation in the undertaking that would eventually spawn the free software movement, Stallman declared, "I'm asking individuals for donations of programs and work," explaining that, while "for most projects, such part-time distributed work would be difficult to coordinate" because "the independently-written parts would not work together," "for the particular task of implementing Unix, this problem is absent." He went on to note that, since "[a] complete Unix system contains hundreds of utility programs, each of which is documented separately" and "[m]ost interface specifications are fixed by Unix compatibility," if "each contributor can write a compatible replacement for a single Unix utility, and make it work properly in place of the original on a Unix system; then these utilities will work right when put together." What separated the task of implementing Unix from other projects to which this distributed model of development might not be so applicable, then, was the modularity inherent in the Unix design. Thanks to this modularity, each utility could be implemented independently by programmers working independently and could be expected to work when incorporated into GNU as a whole.<sup>75</sup>

As the project progressed, programmers working on it also took steps to ensure that individual utilities had a modular design, often expressly because doing so would make it easier for future programmers to contribute to those utilities. For example, while discussing design

---

<sup>75</sup> Richard Stallman, "The GNU Manifesto," in *Free Software, Free Society: Selected Essays of Richard M. Stallman*, ed. Joshua Gay (Boston: GNU Press, 2002), 35.

choices that would need to be made while writing a window manager for the GNUStep programming environment in a GNU-oriented Usenet newsgroup, one programmer wrote, “I tend to think that modularity is the key for expansion. So perhaps the window manager could fit in such that things people might need to/want to change could easily be changed.”<sup>76</sup> Another programmer, while discussing a potential redesign of the Concurrent Versions System (CVS), a free program for managing revisions to pieces of software, had as his first concern the notion that “CVS shouldn’t become too large, [...] a project management [and] build tool shouldn’t be integrated directly into CVS. CVS is a source code control tool. If the number of commands is increased more and more, it will become too complicated, too hard to test and, especially for beginners, too hard to learn.”<sup>77</sup> Instead, this programmer proposed, “The code of CVS should be divided into several modules or libraries which are independent from each other,” listing the advantages of this approach as being that the “modules could easily be reused by other programs and graphical user interfaces [...] or even project management and build tools. Their programmers don’t have to invent the wheel again,” that by “putting together all the modules a more powerful tool can be built step by step. Developers don’t have to use all the features and capabilities from the beginning and they don’t have to learn about functions, which they probably would never need,” and that “splitting up the functionality into separate modules has a lot of advantages during testing.”<sup>78</sup> Making the CVS code modular, in other words, would make it possible for programmers to use its modules again in other projects and to make and test adjustments to individual modules independently of the others.

---

<sup>76</sup> Prashant Singh [prash@ghg.net], “RE:window manager” in [gnu.gnustep.discuss], 20 September 1998, <http://groups.google.com/group/gnu.gnustep.discuss>.

<sup>77</sup> Lars-Christian Schulze [schulze@aerodata.de], “RE:Again: CVS redesign” in [gnu.cvs.help], 14 October 1999, <http://groups.google.com/group/gnu.cvs.help>.

<sup>78</sup> *Ibid.*

By the early 1990s, according to Stallman, the GNU operating system was nearly complete. Almost all of the system's utility programs had been written, and all that was left to write was the Hurd, the GNU kernel. The programmers working on GNU had decided to implement the Hurd "as a collection of server processes," or, to explain the name, a "herd of gnus" that would run on top of another kernel, Mach, and "do the various jobs of the Unix kernel," with the result that "the start of development was delayed as we waited for Mach to be released as free software, as had been promised."<sup>79</sup> Difficulties in the debugging process further stalled the Hurd's development, and in 1991, long before the Hurd was ready for release, a twenty-one year old Finnish programmer named Linus Torvalds began and completed preliminary work on another free software project: the development of a Unix-based kernel called Linux. Torvalds initially described Linux as "just a hobby" and said that it "won't be big and professional like gnu."<sup>80</sup> As more people began contributing to Linux, though, it advanced, and by 1992 it could be combined with the GNU utilities to form a complete, free, Unix-like operating system known as GNU/Linux.

Among members of the community surrounding Linux, ideas similar to Stallman's regarding the implications of a modular design for the distributed development of free software were in circulation. For example, in one post, a programmer working on Linux defended its design from an accusation of insufficient modularity by writing, "This isn't true—although the kernel is one big chunk of code, it is subdivided into smaller modules, such as the VM code, each one of the file systems, each device driver, etc.," and, though he admitted that "Some of these modules have incestuous relationships with each other due to various kludges and hacks"

---

<sup>79</sup> Richard Stallman, "The GNU Project," in *Free Software, Free Society: Selected Essays of Richard M. Stallman*, ed. Joshua Gay (Boston: GNU Press, 2002), 27.

<sup>80</sup> Linus Torvalds [torvalds@klaava.Helsinki.FI], "What would you like to see most in minix?" in [comp.os.minix], 26 August 1991, <http://groups.google.com/group/comp.os.minix/topics>.

and that, to modify the kernel, “[y]ou do have to understand some basic ‘common’ things used” in it, he insisted that “to say you have to understand the whole kernel to write a device driver is a gross overstatement.”<sup>81</sup> That he felt the need to defend Linux on the grounds of the ease with which programmers would be able to change it shows that he considered this to be an important criterion, and the way he did so shows that he viewed modularity as intricately tied to this criterion. A later post announcing the release of a more “modularized” version of the Linux kernel, which opened with the declaration that, thanks to the modularization of the kernel, “the efforts of individual working groups need no longer affect the development of the kernel proper,” likewise indicates the presence of an attitude toward the importance of modularity to the development of free software much like Stallman’s.<sup>82</sup> Moreover, it indicates that programmers working on Linux regarded the modularity of a system as an important enough factor in its maintainability to expend the effort needed to modularize it.

Given the Linux community’s receptive attitude toward the potential for modularity to facilitate the radically distributed development of software, it is perhaps unsurprising that Linux progressed as rapidly as it did from Torvalds’ initial release. Yet Linux was a single program, unlike the GNU utilities, and, although it had a fairly-modular design, it also had a number of “kludges and hacks” that likely caused some of its modules to be more tightly coupled than they could have been. Why, then, was Linux completed so quickly, while the GNU Hurd, the fundamental design of which was much more aggressively modular, has yet to be released and, given the dominance of GNU/Linux in the “market” for free and open source operating systems, may never be completed?

---

<sup>81</sup> Drew Eckhardt [drew@ophelia.cs.colorado.edu], “RE:GNU kids on the block? (sorry... couldn’t resist)” in [comp.os.linux], 29 August 1992, <http://groups.google.com/group/comp.os.linux>.

<sup>82</sup> Peter MacDonald [pmacdona@sanjuan.uvic.ca], “RE:SLS 1.05: Softlandings Modular Linux Released,” in [comp.os.linux.admin], 4 April 1994, <http://groups.google.com/group/comp.os.linux.admin/topics>.



If Eric S. Raymond, co-founder of the Open Source Initiative, co-developer of multiple free software projects, and author of a number of influential papers on the development of free and open source software, is to be believed, it is because Torvalds and the community surrounding Linux took full advantage of their software's modularity—something the GNU Project of the early 1990s failed to do. In *The Cathedral and the Bazaar*, the Boston-born programmer Raymond—who had previously been known for his work on NetHack, a famous and free computer role-playing game—sought to understand how it was that “a world-class operating system could coalesce as if by magic out of part-time hacking by several thousand developers scattered all over the planet, connected only by the tenuous strands of the Internet.”<sup>83</sup> He explained that, prior to the advent of Linux, he “had been preaching the Unix gospel of small tools, rapid prototyping and evolutionary programming for years,” but he “also believed there was a certain critical complexity above which a more centralized, *a priori* approach was required,” that “the most important software (operating systems and really large tools like the [GNU] Emacs programming editor) needed to be built like cathedrals, carefully crafted by [...] small bands of mages working in splendid isolation, with no beta to be released before its time.”<sup>84</sup> Richard Stallman held the same view in 1985, when, after proposing that the tasks of writing and testing the various GNU utilities be taken on by individual programmers working independently, he promptly noted that the “kernel will require closer communication and will be worked on by a small, tight group.”<sup>85</sup> This belief was also reflected in the methodology used for writing individual GNU utilities. Though the work of writing the entire set of utilities was distributed among many programmers, Raymond noted, “the Emacs C core and most other GNU

---

<sup>83</sup> Eric S. Raymond, *The Cathedral and the Bazaar: Musings on Linux and Open-Source by an Accidental Revolutionary* (Cambridge: O'Reilly, 1999), 29.

<sup>84</sup> *Ibid.*

<sup>85</sup> Stallman, “The GNU Manifesto,” 35.

tools” were developed using a “cathedral-building style,” in which a small group of programmers wrote and tested the code in relative isolation, “only releas[ing] a version every six months (or less often),” when the new version seemed stable enough for widespread use, “and work[ing] like a dog on debugging between releases.”<sup>86</sup>

Torvalds took a different approach. His style was to “release early and often, delegate everything you can, [and] be open to the point of promiscuity,” so that, rather than being characterized by “quiet, reverent cathedral building,” “the Linux community seemed to resemble a great babbling bazaar of different agendas and approaches (aptly symbolized by the Linux archive sites, which would take submissions from *anyone*).” In a manner that initially shocked Raymond, the project “not only didn’t fly apart in confusion, but seemed to go from strength to strength at a speed barely imaginable to cathedral builders,” that is, Stallman and the GNU Project. After successfully leading development on a free email-retrieval client, “Fetchmail,” in a manner that was consciously chosen to imitate Torvalds’ approach, Raymond came to the conclusion that the Linux model worked as well as it did because it effectively leverage its users, many of whom were programmers themselves, to “diagnose problems, suggest fixes, and help improve the code far more quickly than [Torvalds] could unaided,” something made possible both by a rapid release schedule that kept users interested in contributing by regularly incorporating these user-produced modifications into the “official” Linux kernel and by the fact that, as Raymond put it, “[g]iven enough eyeballs, all bugs are shallow” and the corresponding fixes easy for someone. Herein, according to Raymond, lay the key to Linux’s success, the crucial difference between the Linux and early-1990s GNU models of free software development: in the eyes of cathedral-builders, “bugs and development problems are tricky, insidious, deep phenomena. It takes months of scrutiny by a dedicated few to develop confidence

---

<sup>86</sup> Raymond, *The Cathedral and the Bazaar*, 37-38.

that you've winkled them all out. Thus the long release intervals, and the inevitable disappointment when long-awaited releases are not perfect," whereas, in the bazaar approach, "you assume that bugs are generally shallow phenomena—or, at least, that they turn shallow quickly when exposed to a thousand eager co-developers pounding on every new release. Accordingly you release often in order to get more corrections, and as a beneficial side effect you have less to lose if an occasional botch gets out the door." By making the process of software development as open as its end result, Linux was able to invite the participation of thousands of programmers in that process and, consequently, to truly live up to the potential for widely distributed development granted by its modularity.<sup>87</sup>

From its inception with Richard Stallman's "The GNU Manifesto," the free software movement recognized modular programming's potential application as a tool to enable programmers to collaborate on complex software projects by breaking them down to be distributed among those working on them. The GNU Project put this to use by distributing the development of individual Unix-like utility programs among volunteers, but, because it assumed that each utility—along with the kernel itself—needed to be developed by an individual or a small group of "cathedral builders," it failed to take full advantage of modularity's benefits with regard to the development of individual programs. It was only when the Linux kernel's development process rapidly overtook that of the GNU Hurd thanks to Linus Torvalds' policy of making frequent releases of the kernel and openly accepting contributions from its users that these benefits became evident. Partially thanks to Eric Raymond's criticism of the GNU Project, the burgeoning free software movement took notice of these benefits, with many projects adopting or switching to what Raymond would call a "bazaar" development model. However,

---

<sup>87</sup> *Ibid.*, 30, 36, 41-42.

Raymond's criticism also foreshadowed a division that would soon split this community, beginning in the late 1990s.

### **Pragmatism and the Permissibility of Proprietary Software**

Eric Raymond's examination and criticism of the GNU Project's development model reflected his priorities. Broadly speaking, he was interested primarily in the pragmatic implications of free software. He believed that, if approached in the proper manner, it could be developed more efficiently than proprietary software, with better end products as the result. Richard Stallman and the Free Software Foundation were not entirely indifferent to Raymond's concerns. As Raymond himself noted, by the late 1990s, many GNU utilities—including the GNU C Compiler, the initial release of which Stallman himself had been the primary developer—had shifted to the bazaar model, either by making use of a rapid release cycle like that of early Linux or by making incremental changes to the source code publicly visible even between releases.<sup>88</sup> However, their priorities lay elsewhere. They saw the distributed development model used in producing free software more as a way to combat what it saw as infringements on programmers' liberty than as a way to produce better programs more quickly. During the decade following the development of the Linux kernel, Raymond and other programmers who shared his views grew increasingly frustrated with the perceived zealotry and anti-commercialism of the Free Software Foundation, to the point that, by 1998, they were eschewing the term “free software” in favor of the term “open source” and setting up a parallel organization to the Free Software Foundation called the Open Source Initiative. The two camps into which the free software movement had become divided were delineated by their positions on one issue in particular: the morality of proprietary software.

---

<sup>88</sup> Raymond, *The Cathedral and the Bazaar*, 258.

To Stallman and the other partisans of free, rather than open source, software, restricting users' ability to modify and redistribute software was simply reprehensible. To quote "The GNU Manifesto," "[t]here is nothing wrong with wanting pay for work, or seeking to maximize one's income, as long as one does not use means that are destructive," but "[e]xtracting money from users of a program by restricting their use of it is destructive because the restrictions reduce [...] the amount of wealth that humanity derives from the program." According to Stallman, "a good citizen does not use such destructive means to become wealthier" because "if everyone did so, we would all become poorer from the mutual destructiveness." Here, Stallman argued from the Kantian perspective that making software proprietary violates the categorical imperative. Earlier, he had argued from the utilitarian perspective that, for programmers, to whom "[c]opying all or parts of a program is as natural [...] as breathing," a society that arranged to cover the cost of program production through licensing arrangements and intellectual property laws would be analogous to "a space station where air must be manufactured at great cost" that covered said cost by "charging each breather per liter of air." Doing so "may be fair," but "wearing the metered gas mask all day and all night is intolerable even if everyone can afford to pay the air bill" and "the TV cameras everywhere to see if you ever take the mask off are outrageous." He concluded that the better solution would be "to support the air plant with a head tax and chuck the masks." Finally, he outright rejected the notion that "people have a right to control how their creativity is used," saying that "[c]ontrol over the use of one's ideas' really constitute control over other people's lives" and that, far from being intrinsic, "intellectual property rights are just licenses granted by society because it thought, rightly or wrongly, that society as a whole would benefit by granting them."<sup>89</sup>

---

<sup>89</sup> Stallman, "The GNU Manifesto," 36, 38-39.

Others in the free software movement shared Stallman's views on the morality of proprietary software, going as far as to claim that, on the subject of free software, "there is no neutral position." To decide to "copyleft" one's code using the GNU General Public License was to decide to "support the users against the proprietary software makers," while to "reject the GPL" was to decide to do "the opposite." For these programmers, "copylefting" their code to ensure that not only it, but also all programs derived from it, would remain free was an ethical imperative. Doing otherwise meant either directly infringing on users' liberty or failing to prevent future creators of proprietary software derived from the original freely-distributable software from doing so.<sup>90</sup>

Raymond, on the other hand, aligned himself with what he called the more "pragmatist," "less confrontational and more market-friendly strand in hacker culture," which he contrasted with the "more purist and fanatical elements" such as the "very zealous and very anticommercial" Free Software Foundation, with its "vigorous and explicit drive to 'Stamp Out Software Hoarding!'" and its partisans' tendency toward (in Raymond's view) characteristic statements such as "Commercial software is theft and hoarding. I write free software to end this evil." The typical pragmatist, on the other hand, "values having good tools and toys more than he likes commercialism, and may use high-quality commercial software without ideological discomfort," regards "the GPL [...] not as a weapon against 'hoarding,' but as a tool for encouraging software sharing and the growth of bazaar-mode development communities," and has an attitude that "is only moderately anticommercial, and its major grievance against the corporate world is not 'hoarding' per se; rather it is that world's perverse refusal to adopt superior approaches incorporating Unix and open standards and open-source software. If the

---

<sup>90</sup> Alan Curry [pacman@defiant.cqc.com], "RE:Free software vs. open source?" in [gnu.misc.discuss], 12 June 2000, <http://groups.google.com/group/gnu.misc.discuss>.

pragmatist hates anything, it is less likely to be ‘hoarders’ in general than the current King Log of the software establishment—formerly IBM, now Microsoft.” In these quotes, Raymond differentiated his own, “pragmatist” point of view with that of the “zealous” proponents of free software mainly through the two sides’ respective attitudes toward “software ‘hoarding,’” that is, through views on the legitimacy or illegitimacy of intellectual property in software.<sup>91</sup>

Unsurprisingly, given that Raymond was one of the major partisans of the shift toward the use of the term “open source” in lieu of the term “free software,” Raymond’s opinions were shared by many who preferred “open source.” One such programmer, posting on Usenet, responded to a programmer who had just declared that the key differences between the open source movement and the free software movement were “more philosophical and political than technical” with the retort, “Since the biggest problem with the free software movement is RMS’ [Richard Stallman’s] radical leftist politics, this is entirely appropriate.”<sup>92</sup> Another programmer, who declared open source “a better term, IMO, than free software,” suggested that the difference between the two movements was “a bit technical as well; RMS (and thus the FSF) considers that any ‘free license’ that does not permit GPLing isn’t ‘free’ enough. The ‘open source’ folks [...] consider open source to be source-available with unrestricted redistribution for changes to the source (at a minimum!) but do not generally require that derivatives be exclusively source-available (only those parts that are from or within the original ‘open source’),” with the result that while open source “allows you to make commercial products that are a mix of open- and closed-source, the other does not,” enabling open source and proprietary software to coexist more easily than free and proprietary software.<sup>93</sup> Raymond and the partisans of open source,

---

<sup>91</sup> Raymond, *The Cathedral and the Bazaar*, 82-86.

<sup>92</sup> Jay Maynard [jaymaynard@thebrain.conmicro.cx], “RE:Free software vs. open source?” in [gnu.misc.discuss], 9 June 2000, <http://groups.google.com/group/gnu.misc.discuss>.

<sup>93</sup> Austin Ziegler [aziegler@the-wire.com], “RE:Free software vs. open source?” in [gnu.misc.discuss], 13 June

then, did not regard the use of a closed, proprietary model for the development and distribution of software as unethical so much as inefficient, and this was reflected in their choice of the term “open source.”

Perhaps the most striking evidence of the rift between the open source movement and the free software movement appears in a 1999 Usenet post in which the computer programmer and author of the “Open Source Definition” Bruce Perens announced his resignation from the board of the Open Source Initiative, which he had cofounded with Raymond the previous year. On the one hand, Perens did not believe that the rift had arisen from fundamental differences within the free and open source software community. As he declared, “Eric Raymond and I founded the Open Source Initiative as a way of introducing the non-hacker world to Free Software” and “although some disapprove of Richard Stallman’s rhetoric and disagree with his belief that all software should be free, the Open Source Definition is entirely compatible with the Free Software Foundation’s goals, and a schism between the two groups should never have been allowed to develop.” Nevertheless, Perens noted, “I fear that the Open Source Initiative is drifting away from the Free Software values with which we originally created it,” with the result that “The Open Source certification mark has already been abused in ways I find unconscionable and that I will not abide.” Saying that “We must make it clear that those freedoms [associated with free software] are still important, and that software such as Linux would not be around without them,” he announced that he would continue to work to promote these freedoms “independently from the Open Source Initiative.” Despite seeing the division between the advocates of open source and those of free software as ultimately unnecessary, Perens felt that



the Open Source Initiative had departed sufficiently from its roots in the free software movement to necessitate his resignation.<sup>94</sup>

Divergent priorities and beliefs about the morality of proprietary software gave rise to divisions among programmers working on freely-distributable software, with the community splitting into those that preferred the term “free software” and those that preferred the term “open source software.” In the eyes of the advocates of open source, the advocates of free software were zealous and anti-commercialist to a degree that interfered with their ability to truly realize the pragmatic benefits of their development model, while in the eyes of the advocates of free software, the advocates of open source were insufficiently committed to the goal of liberating programmers from proprietary restrictions on software. That said, it is easy to overstate the extent of the division between the two sides. Despite their disagreements, the partisans of free and open source software formed a community of programmers with both shared goals and a shared vision of freely-distributed software’s potential to transform the software development industry and, eventually, the rest of society.

### **Revolutionaries, by Intent and by Accident**

By the late 1990s, the insistence of the partisans of free software on the moral bankruptcy of proprietary software had divided them from the partisans of open source. Nevertheless, the open source movement grew out of the free software movement, and the goals, ideals, and mores of the two remained closely linked. Members of the free software movement were not uncompromisingly anti-commercial, and members of the open source movement were not strictly pro-market. The definitions of free software and open source software were similar enough to ensure that virtually every program that was considered “open source” would also be

---

<sup>94</sup> Bruce Perens [bruce@k6bp.hams.com], “RE:It’s Time to Talk About Free Software Again” in [muc.lists.debian.user], 18 February 1999, <http://groups.google.com/group/muc.lists.debian.user>.

considered “free” and vice versa. The partisans of open source, like those of free software, believed that their efforts would benefit the community of programmers as a whole. Perhaps most significantly, both groups embraced the idea that their communities—and the connected social structures that encouraged the free exchange of information and facilitated a production model based on voluntary cooperation—could provide a template for the transformation of society into one free of the hierarchical institutions characteristic of mainstream society.

Apart from their disagreements regarding the ethical status of proprietary software, the positions of Stallman’s Free Software Foundation on the one hand and Raymond’s Open Source Initiative on the other were remarkably similar. In *The Cathedral and the Bazaar*, Raymond overstated the extent to which the Free Software Foundation’s opposition to intellectual property in software was tied to an opposition to commercialism in general that contrasted with the pragmatists’ more pro-market attitudes. Stallman himself had proposed ways in which companies might profit from the existence of GNU, writing, “If people would rather pay for GNU plus service than get GNU free without service, a company to provide just service to people who have obtained GNU free ought to be profitable,” and, indeed, this is precisely how companies such as Red Hat, which sells support to users of otherwise-free software, would come to operate.<sup>95</sup> Raymond, meanwhile, despite his pronounced libertarian leanings, was hardly a market fundamentalist. While he claimed that the “verdict of history seems to be that free-market capitalism is the globally optimal way to cooperate for economic efficiency” in a context of scarce resources, he also claimed that “perhaps, in a similar way, the reputation-game gift culture is the globally optimal way to cooperate for generating (and checking!) high-quality creative work” in a context lacking scarcity.<sup>96</sup> Raymond described the free and open source software

---

<sup>95</sup> Stallman, “The GNU Manifesto,” 37.

<sup>96</sup> Raymond, *The Cathedral and the Bazaar*, 132.

community as such a “reputation-game gift culture,” in which people are motivated principally by the desire to improve their reputation by producing gifts to freely distribute to others, and indicated that it represented an concrete example of how a “severe effort of many converging wills” could be achieved through the “principle of common understanding” as “vaguely suggested” by the Russian anarchist and communist Pyotr Kropotkin.<sup>97</sup> The free and open source software community, Raymond suggested, resembled the sort of stateless society that Kropotkin had advocated, organized as it was according to the principle “to every man according to his needs” through “free contacts between individuals and groups pursuing the same aim.”<sup>98</sup> Given this, it is perhaps unsurprising that one programmer, while describing the culture of the open source movement, declared that “the fundamental belief behind open source software is that ‘Information Should be Free,’” that “the open source movement is a very communist, a very communal, activity; as such it doesn’t really support capitalist values very well,” and that those who “hope to draw off of its resources without giving back to the community” should “return to the Cathedral [...] back to your little world of proprietary software and suits.”<sup>99</sup>

The ideas of the open source and free software movements regarding what constituted open source or free software respectively likewise remained closely connected. When describing his objections to the use of the term “open source” as opposed to “free software,” Stallman noted that the “official definition of ‘open source software,’ as published by the Open Source Initiative, is very close to our definition of free software,” in particular requiring both that such software be freely modifiable and redistributable and that any software derived from open source code also be open source. However, he also noted that “the obvious meaning for the expression ‘open

---

<sup>97</sup> *Ibid.*, 63-64.

<sup>98</sup> Pyotr Kropotkin, *The Conquest of Bread*, Paris (1892), reprint, New York: Vanguard Press (1926), 26, 29.

<sup>99</sup> Levi [levi@top.monad.net], “RE:Open Source Development” in [comp.os.research], 5 August 1998, <http://groups.google.com/group/comp.os.research>.

source software' is 'You can look at the source code,'" which "is a much weaker criterion."

Despite the inexact nature of the term "open source," then, its official definition reflected the fact that the open source movement effectively spun out of the free software movement as much as it reflected the open source movement's desire to make itself more palatable to industry.<sup>100</sup>

The partisans of open source and of free software remained connected on a deeper level, as well: both, perhaps unsurprisingly, saw the proliferation of free and open source software as something that would serve the best interests of programmers as a group. Stallman's first concerns were for the autonomy and camaraderie of programmers. As he wrote when describing his reasons for initiating the GNU Project, "Many programmers are unhappy about the commercialization of system software. It may enable them to make more money, but it requires them to feel in conflict with other programmers in general rather than feel as comrades. The fundamental act of friendship among programmers is the sharing of programs; marketing arrangements now typically used essentially forbid programmers to treat others as friends."<sup>101</sup> However, he was not dismissive of programmers' need for an income as well, nor of the implications a society without intellectual property would have for their ability to do so. On this point, he cited his own experience that "[p]rogrammers writing free software can make their living by selling services related to the software. I have been hired to port the GNU C compiler to new hardware, and to make user-interface extensions to GNU Emacs [...] I also teach classes for which I am paid" and that even if independent proprietary software vendors were no longer viable in such a society, hardware manufacturers would still "find it essential to support software development even if they cannot control the use of the software. In 1970, much of their software was free because they did not consider restricting it. Today, their increasing willingness to join

---

<sup>100</sup> Richard Stallman, "Why 'Free Software' is Better than 'Open Source,'" in *Free Software, Free Society: Selected Essays of Richard M. Stallman*, ed. Joshua Gay (Boston: GNU Press, 2002), 58.

<sup>101</sup> Stallman, "The GNU Manifesto," 35.

consortiums show their realization that owning the software is not what is really important for them.”<sup>102</sup>

Similarly, Raymond addressed the argument that a shift from proprietary to free and open-source software would devalue programmers’ labor by claiming that, practically speaking, “most programmer-hours are spent (and most programmer salaries are paid for) writing or maintaining in-house code that has no sale value at all,” from “the financial- and database- software customizations every medium and large company needs” to “all kinds of embedded code for our increasingly microchip-driven machines—from machine tools and jet airliners to microwave ovens and toasters,” with the result that “software is largely a service industry operating under the persistent but unfounded delusion that it is a manufacturing industry.”<sup>103</sup> A transition to open source would not, he argued, eliminate the need for this sort of programming work. Programmers’ salaries would still get paid. However, the community of programmers as a whole would also enjoy the advantages of the presence of “[m]ore and more high-quality software [...] permanently available to use and build on instead of being discontinued or locked in somebody’s vault.”<sup>104</sup> Finally, when describing the motivations of the programmers who worked on free and open source projects, one such programmer declared that “much of the work done by hackers on open source software is for either of two reasons: a) gratitude for the software being free, or b) a sort of mystical ‘giving back’ to the \*whole\* free software community,” motivations whose very existence indicate that these programmers felt that free and open source software represented a net gain for the software development community.<sup>105</sup>

---

<sup>102</sup> Richard Stallman, “Why Software Should Be Free,” in *Free Software, Free Society: Selected Essays of Richard M. Stallman*, ed. Joshua Gay (Boston: GNU Press, 2002), 121.

<sup>103</sup> Raymond, *The Cathedral and the Bazaar*, 142-143, 145.

<sup>104</sup> *Ibid.*, 194.

<sup>105</sup> Levi [levi@top.monad.net], “RE:Open Source Development” in [comp.os.research], 5 August 1998, <http://groups.google.com/group/comp.os.research>.

Somewhat more surprisingly, given the open source proponents' supposed "pragmatism," both they and the partisans of free software showed signs of seeing free and open source software as closely linked to broader political and social goals. Stallman declared, for example, that "making programs free is a step toward the post-scarcity world, where nobody will have to work very hard just to make a living" and that "we have already greatly reduced the amount of work that the whole society must do for its actual productivity, but only a little of this has translated itself into leisure for workers because much nonproductive activity is required to accompany productive activity," adding that "[f]ree software will greatly reduce these drains in the area of software production."<sup>106</sup> After resigning from the Open Source Initiative's board, Bruce Perens expended considerable effort toward "expanding the scope of collaborative works beyond software," particularly in the area of the dissemination of knowledge.<sup>107</sup> As the editor of a series of freely-distributable books on open source software published by Prentice Hall, he expressed the hope that electronic versions of such books might be frequently updated in accordance with feedback from readers in a manner reminiscent of the maintenance of an open source software project.<sup>108</sup> Raymond was more cautious than either Stallman or Perens in his observations regarding the social implications of open source, saying that "staying focused on the goal" of advancing open source software itself and "not wandering down a lot of beguiling byways" was particularly important for the open source movement "because in the past our representatives have shown a strong tendency to ideologize when they would have been more effective sticking to relatively narrow, pragmatic arguments," referring, of course, to Stallman.<sup>109</sup> However, he also wrote that "the success of open source does call into some question the utility

---

<sup>106</sup> Stallman, "The GNU Manifesto," 41.

<sup>107</sup> Steve Lohr, "Steal This Book? A Publisher Is Making It Easy," *The New York Times*, 13 January 2003, Technology section.

<sup>108</sup> *Ibid.*

<sup>109</sup> Raymond, *The Cathedral and the Bazaar*, 226.

of command-and-control systems, of secrecy, of centralization, and of certain kinds of intellectual property,” that “it suggests (or at least harmonizes well with) a broadly libertarian view of the proper relationship between individuals and institutions,” and that “I expect the open-source movement to have essentially won its point about software within three to five years. Once that is accomplished [...] it will become more appropriate to try to leverage open-source insights in wider domains. In the meantime, even if we hackers are not making an ideological noise about it, we will still be changing the world.”<sup>110</sup>

Real differences existed between the programmers in the free and open source software movements who gravitated toward the term “open source” and those who kept to the term “free,” particularly with regard to the question of the legitimacy of intellectual property in software and, in general, with regard to what the partisans of open source saw as the free software movement’s overly-confrontational attitude. However, ultimately, the two factions were more alike than not. Their shared values and goals indicate that each comprised a subset of a community of programmers who embraced the idea that freely-distributed software was either ethically or practically superior to proprietary software and, further, saw in the process of developing such software a model for an alternative social order in which hierarchical “command-and-control” systems might be replaced with systems driven by the autonomous cooperation of individuals. Eric Raymond may have considered himself an “accidental revolutionary” and, in contrast to Richard Stallman’s emphasis of his belief that the spread of free software was a step on the path to a post-scarcity world, regarded the open source development model’s social implications as a low priority consideration when compared with that model’s implications for the quality of software and the efficiency of the software production process. Yet the pragmatism characteristic of the open source movement lay atop a foundation of idealism embodied in Raymond’s

---

<sup>110</sup> *Ibid.*, 227.

assurance that, even when stripped of “ideological noise,” the open source movement’s efforts were “changing the world.”

\* \* \* \* \*

During the late 1960s, the New Communalists had sought to apply technology to assert their autonomy and to build alternative communities that would serve as templates for the transformation of mainstream society. Two decades later, a community of programmers that had internalized a set of values rooted in the counterculture applied modular programming techniques to much the same end, giving rise to the free software movement. The opinions of the programmers involved in this movement were far from unanimous, particularly with regard to the permissibility of proprietary restrictions on software development, and, as a result, a subset of the movement spun off in the late 1990s, referring to itself as the “open source” movement. Both the free software movement and the open source movement made use of modularity to enable the undertaking of enormous projects in which they were able to act autonomously, but cooperatively. Both saw their actions as being in the best interests of programmers as a group, a way to counteract the negative effects of the increasingly-stringent intellectual property laws on these programmers’ ability to share information and to build on the knowledge of other programmers by creating a large base of freely-available and freely-adaptable code. Finally, both envisioned their projects—and the underlying development model, which relied on the voluntary cooperation of large numbers of programmers to make them possible—as exemplars of alternative principles of social organization, which might provide a basis for the transformation of society at large according to these principles. Modular programming, conceived and propagated against a countercultural backdrop, was applied to build a community of programmers who, like the New Communalists before them, came to see themselves as potential



heralds of revolutionary social change that would be driven not by political action, but rather by example. By showing that enormous, complex projects could be completed by programmers cooperating on an entirely autonomous basis, they would not only provide a way for programmers to continue freely sharing and adapting code as they had prior to the developments in intellectual property law of the early 1980s. They would also plant the seeds of broader social transformation by, to paraphrase Eric Raymond, calling into question the desirability of secrecy, of centralization, and of hierarchical social structures in general.

## Conclusion

When Stewart Brand gathered over a hundred prominent programmers together for the Hacker's Conference of 1984, he did so hoping that the "hacker" community had the potential to form "a precursor to a larger culture," a vanguard that would initiate the kinds of social transformations that members of the 1960s counterculture—in particular, the New Communalists—had sought to bring about.<sup>111</sup> The decades following the free software movement's emergence from this community vindicated Brand's hopes. Free and open source software came to be used extensively throughout the software development industry, to the point that many proprietary software vendors began to use free tools during development or even to incorporate open source components in their products. Thus, by 2008, Bruce Perens could claim that "Free Software / Open Source is mainstream," citing its "pervasive penetration [...] in business servers and embedded systems," its entrenchment in the market for network server software, and "the fact that Free Software provides a large part of Apple's MacOS today, and critical elements of Microsoft Windows as well."<sup>112</sup>

More significant than the widespread adoption of free software itself, though, was the growth of an Internet culture infused with the free software community's values and methodologies. In 2003, Perens had experimented with the idea of applying open source development principles to the compilation of knowledge while editing Prentice Hall's book series on open source software, but perhaps the most radical—and successful—such application of these principles had already been launched two years previously: Wikipedia. Dubbing itself "The Free Encyclopedia," Wikipedia relied on its users to voluntarily collaborate on the creation and improvement of articles much as the free and open source software movement relied on the

---

<sup>111</sup> Turner, *From Counterculture to Cyberculture*, 136.

<sup>112</sup> Bruce Perens, "State of Open Source Message: A New Decade for Open Source," 8 February 2008, <http://perens.com/works/articles/State8Feb2008.html> (accessed 25 March 2013).

users of software to voluntarily collaborate on its production and improvement. Alongside the proliferation of websites reliant on such user-generated content, organizations such as Creative Commons sought to provide a way to keep such content free by generalizing the GNU General Public License's notion of "copyleft" to apply to works other than software. To this end, Creative Commons—two of whose co-founders, Lawrence Lessig and Hal Abelson, had been or would be members of the Free Software Foundation's board of directors—produced a set of licenses designed to permit the free redistribution of creative works, some of which, like the GPL, also required modified or "derived" works to be distributed under the same license. Like the community of programmers that gave rise to the free software movement, the digital communities that used these licenses to permit the free distribution and derivation of creative works demonstrated a commitment to the value of voluntary collaboration and adaptation, and, in particular, to the idea that information should be free. The New Communalists and, perhaps to a lesser extent, the computer scientists of the late 1960s had regarded themselves as elites, as cultural vanguards who would lead society at large through transformations: in the case of the New Communalists, into a new society organized along networked, collaborative, nonhierarchical lines that would preserve individual autonomy, and in the case of the computer scientists, into a new society in which "the art of programming" would be "public property," as Andrei Ershov put it, much like reading and writing. By instilling the emerging culture of the Internet with their values and methods, the free software movement played the role of a cultural vanguard to a degree greater than either of the two groups preceding it had achieved. In the process, the methods the programming experts in this movement applied to achieve autonomy for themselves generalized, providing ways for all interested Internet users to autonomously collaborate on, share, and modify creative works, from encyclopedia articles to videos.

By applying computer and computer network technology to build communities founded on the principle of voluntary cooperation, these Internet users took technological products of government bureaucracies and turned them into tools, much as the New Communalists had attempted to do during the late 1960s. This mirrored the manner in which programmers had sought to apply modular programming since formulating it contemporaneously with the New Communalists' efforts to transform society. Initially, computer scientists had conceived of modular programming as a means to provide the base of common knowledge and standards programmers would need to gain the status of professionals and secure their autonomy in the face of managerial efforts to control them. Their professionalization efforts did not succeed. However, by the early 1980s, these computer scientists had managed to propagate both the techniques associated with modular programming and the notion that modularity was a characteristic of "good" code throughout the broader community of programmers, particularly those using the modern Internet's predecessor, Usenet, to communicate with each other. These programmers, following utilitarian ideas about aesthetics in regarding utility as a particularly good criterion by which to judge creative works, came to see modular programming as a means by which to make a program more aesthetically pleasing by making it clearer, more readable, and thus more useful to other programmers. By encouraging programmers to regard programming as a worthwhile creative activity in itself and emphasizing the importance of readability and adaptability, this encouraged the growth of a community of programmers who had internalized the counterculturally-influenced "hacker ethic," which held that programs should be freely available to programmers to read, adapt, and redistribute. It was from this community that the free software movement emerged, in response to the threat to programmers' freedom to do just this embodied in proprietary restrictions on software. By applying modular

programming to distribute the task of programming replacements for fundamental pieces of proprietary software among many programmers, the free software movement aimed to restore the freedoms that proprietary restrictions on software distribution had removed. Moreover, it built a community around a system of voluntary collaboration, providing an example of an alternative social structure that the emerging Internet culture would reproduce many times.

Perhaps the managers of the 1960s had seen the modular programming as a potential instrument by which they could control programmers. As programmers formulated, propagated, and applied modular programming as a tool to secure their own autonomy, though, they demonstrated that it, like the Internet and computers themselves, could act instead as an instrument of liberation and social transformation.

## Bibliography

### Primary Sources

Adams, Glenn [glenn@LL-XN.ARPA]. "RE:code quality." In [net.unix-wizards], 9 October 1985. <http://groups.google.com/group/net.unix-wizards/topics>.

Black, Eric [eric@chronon.UUCP]. "RE:lex and yacc in the public domain (responses)." In [net.unix-wizards], 1 May 1986. <http://groups.google.com/group/net.unix-wizards/topics>.

Brownell, Dave [brownell@harvard.ARPA]. "RE:Object oriented (flames at end)." In [net.lang], 21 June 1984. <http://groups.google.com/group/net.lang/topics>.

Crane, John [crane@fortune.UUCP]. "RE:Self-modifying code." In [net.lang], 3 January 1984. <http://groups.google.com/group/net.lang/topics>.

Crawford, Matt [matt@odjjob.UChicago.UUCP]. "RE:The realest programmer of all." In [net.jokes], 20 November 1984. <http://groups.google.com/group/net.jokes/topics>.

Crispin, Mark [Admin.MRC@SU-SCORE.ARPA]. "RE:public domain?" In [net.unix-wizards], 26 December 1984. <http://groups.google.com/group/net.unix-wizards/topics>.

Curry, Alan [pacman@defiant.cqc.com]. "RE:Free software vs. open source?" In [gnu.misc.discuss], 12 June 2000. <http://groups.google.com/group/gnu.misc.discuss>.

Dijkstra, Edsger. "The Humble Programmer." *Communications of the ACM* 15, no. 10 (1972): 859-866. doi:10.1145/355604.361591.

Dijkstra, Edsger. "Notes on Structured Programming." In *Structured Programming*, edited by Ole-Johan Dahl, Edsger Dijkstra, and C. A. R. Hoare, 1-82. London: Academic Press Ltd., 1972.

Eckhardt, Drew [drew@ophelia.cs.colorado.edu]. "RE:GNU kids on the block? (sorry... couldn't resist)." In [comp.os.linux], 29 August 1992.  
<http://groups.google.com/group/comp.os.linux>.

Ershov, Andrei. "Aesthetics and the Human Factor in Programming." *Communications of the ACM* 15, no. 7 (1972): 501-505. doi:10.1145/361454.361458.

Farber, Dave [farber@udel-huey.ARPA]. "RE:software ethics." In [net.unix-wizards], 3 March 1985. <http://groups.google.com/group/net.unix-wizards/topics>.

Frank, Dan [g-frank@gumby.UUCP]. "RE:High-levelity." In [net.lang], 26 December 1984.  
<http://groups.google.com/group/net.lang/topics>.

Gast, David [gast@ucla-cs.ARPA]. "RE:lex and yacc in public domain (responses)." In [net.unix-wizards], 3 May 1986. <http://groups.google.com/group/net.unix-wizards/topics>.

Gold, Barry [barryg@sdcrcdf.UUCP]. "RE:Where's the (c) on unix?" In [net.unix-wizards], 21 March 1984. <http://groups.google.com/group/net.unix-wizards/topics>.

Gotterbarn, Don. "Software Engineering as a Profession." *ACM SIGSOFT Software Engineering Notes* 23, no. 6 (1998): 205-206. doi:10.1145/291252.295145.

Gould, Ed [ed@mtxinu.UUCP]. "RE:software ethics." In [net.unix-wizards], 21 March 1985.  
<http://groups.google.com/groups/net.unix-wizards/topics>.

Hu, Alan [ajh@sdcsvax.UUCP]. "RE:Difficulty of recursion." In [net.lang], 25 March 1983.  
<http://groups.google.com/group/net.lang/topics>.

Kenig, Mark [cbspt002@abnjh.UUCP]. "RE:Teaching programming –GOTO's, Object Oriented Languages, etc." In [net.lang], 20 June 1984.  
<http://groups.google.com/group/net.lang/topics>.

- Knuth, Donald. "Computer Programming as an Art." *Communications of the ACM* 17, no. 12 (1974): 667-673. doi:10.1145/361604.361612.
- Pyotr Kropotkin. *The Conquest of Bread*. Paris: 1892. Reprint, New York: Vanguard Press, 1926.
- Levi [levi@top.monad.net]. "RE:Open Source Development." In [comp.os.research], 5 August 1998. <http://groups.google.com/group/comp.os.research>.
- Livingston, Jim [jiml@pesnta.UUCP]. "RE:REAL PROGRAMMERS." In [net.jokes], 7 September 1984. <http://groups.google.com/group/net.jokes/topics>.
- Lo, C.J. [cjl@iuvox.UUCP]. "RE:Gotos, tail-recursion." In [net.lang], 20 June 1984. <http://groups.google.com/group/net.lang/topics>.
- Lohr, Steve. "Steal This Book? A Publisher Is Making It Easy." *The New York Times*, 13 January 2003, Technology section.
- MacDonald, Peter [pmacdona@sanjuan.uvic.ca]. "RE:SLS 1.05: Softlandings Modular Linux Released." In [comp.os.linux.admin], 4 April 1994. <http://groups.google.com/group/comp.os.linux.admin/topics>.
- Jay Maynard [jaymaynard@thebrain.conmicro.cx]. "RE:Free software vs. open source?" In [gnu.misc.discuss], 9 June 2000. <http://groups.google.com/group/gnu.misc.discuss>.
- Naur, Peter and Brian Randell. *Software Engineering: Report on a conference sponsored by the NATO Science Committee, Garmisch, Germany, 7th to 11th October 1968*. Brussels: NATO Scientific Affairs Division, 1969.
- Newkirk, Dave [dcn@ixn5h.UUCP]. "RE:Interpreters vs. Compilers – Who wins?" In [net.lang], 23 March 1983. <http://groups.google.com/group/net.lang/topics>.



- Parnas, David. "On the Criteria To Be Used in Decomposing Systems into Modules." *Communications of the ACM* 15, no. 12 (1972): 1053-1058. doi:10.1145/361598.361623.
- Perens, Bruce [bruce@k6bp.hams.com]. "RE:It's Time to Talk About Free Software Again." In [muc.lists.debian.user], 18 February 1999. <http://groups.google.com/group/muc.lists.debian.user>.
- Perens, Bruce. "State of Open Source Message: A New Decade for Open Source." 8 February 2008. <http://perens.com/works/articles/State8Feb2008.html> (accessed 25 March 2013).
- Perlow, Ken [ken@ihuxq.UUCP]. "RE:Professionalizing Programmers, Going Off-line." In [net.lang], 8 June 1984. <http://groups.google.com/group/net.lang/topics>.
- Raymond, Eric S. *The Cathedral and the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary*. Cambridge: O'Reilly, 1999.
- Ritchie, Dennis. "The Evolution of the Unix Time-Sharing System." In *Lecture Notes in Computer Science #79: Language Design and Programming Methodology*. Springer Verlag, 1980.
- Rothery, Brian. *Installing and Managing a Computer*. London: Business Books, 1968.
- Schulze, Lars-Christian [schulze@aerodata.de]. "RE:Again: CVS redesign." In [gnu.cvs.help], 14 October 1999. <http://groups.google.com/group/gnu.cvs.help>.
- Sherouse, George [sherouse@unc.UUCP]. "RE:Unix for physicists (attn:finn)." In [net.physics], 14 June 1984. <http://groups.google.com/group/net.physics/topics>.
- Singh, Prashant [prash@ghg.net]. "RE>window manager." In [gnu.gnustep.discuss], 20 September 1998. <http://groups.google.com/group/gnu.gnustep.discuss>.
- Smith, Gregory [greg@utcsri.UUCP]. "RE:lex and yacc in public domain (responses)." In [net.unix-wizards], 6 May 1986. <http://groups.google.com/gropu/net.unix-wizards/topics>.

- Stallman, Richard. "Free Software Definition." In *Free Software, Free Society: Selected Essays of Richard M. Stallman*, edited by Joshua Gay, 43-45. Boston: GNU Press, 2002.
- Stallman, Richard. "The GNU Manifesto." In *Free Software, Free Society: Selected Essays of Richard M. Stallman*, edited by Joshua Gay, 33-41. Boston: GNU Press, 2002. Originally published in *Dr. Dobbs's Journal of Software Tools* 10, no. 3 (1985): 30-35.
- Stallman, Richard. "The GNU Project." In *Free Software, Free Society: Selected Essays of Richard M. Stallman*, edited by Joshua Gay, 33-41. Boston: GNU Press, 2002. Originally published in Chris DiBona, Sam Ockman, and Mark Stone, eds., *Open Sources: Voices from the Open Source Revolution* (O'Reilly, 1999).
- Stallman, Richard. "Why 'Free Software' is Better than 'Open Source.'" In *Free Software, Free Society: Selected Essays of Richard M. Stallman*, edited by Joshua Gay, 57-62. Boston: GNU Press, 2002. Originally written in 1999.
- Stallman, Richard. "Why Software Should Be Free." In *Free Software, Free Society: Selected Essays of Richard M. Stallman*, edited by Joshua Gay, 57-62. Boston: GNU Press, 2002. Originally written in 1992.
- Steinman, Jans [jans@mako.UUCP]. "RE:Definition of Buzzwords: 'Object-Oriented.'" In [net.lang], 23 January 1985. <http://groups.google.com/group/net.lang/topics>.
- Templeton, Brad [brad@looking.UUCP]. "RE:lex and yacc in the public domain (responses)." In [net.unix-wizards], 7 April 1986. <http://groups.google.com/groups/net.unix-wizards/topics>.
- Torvalds, Linus [torvalds@klaava.Helsinki.FI]. "What would you like to see most in minix?" In [comp.os.minix], 26 August 1991. <http://groups.google.com/group/comp.os.minix/topics>.
- Weinstein, Lauren [lauren@rand-unix.ARPA]. "RE:software ethics." In [net.unix-wizards], 2 March 1985. <http://groups.google.com/group/net.unix-wizards/topics>.

Weinstein, Lauren [lauren@rand-unix.ARPA]. "RE:software ethics." In [net.unix-wizards], 3 March 1985. <http://groups.google.com/group/net.unix-wizards/topics>.

Ziegler, Austin [aziegler@the-wire.com]. "RE:Free software vs. open source?" In [gnu.misc.discuss], 13 June 2000. <http://groups.google.com/group/gnu.misc.discuss>.

## Secondary Sources

Benner, Chris. "Computers in the Wild': Guilds and Next-Generation Unionism in the Information Revolution." In *Uncovering Labour in Information Revolutions, 1750–2000*, edited by Aad Blok and Greg Downey, 180-204. Cambridge: Cambridge University Press, 2003.

Ensmenger, Nathan L. *The Computer Boys Take Over: Computers, Programmers, and the Politics of Technical Expertise*. Cambridge: The MIT Press, 2010.

Friedson, Eliot. *Professionalism: The Third Logic*. Cambridge: Polity Press, 2001.

Graham, Stuart and David Mowery. "Intellectual Property Protection in the U.S. Software Industry." In *Patents in the Knowledge-Based Economy*. Washington, DC: The National Academies Press, 2003. 219-258.

Kraft, Philip. *Programmers and Managers: The Routinization of Computer Programming in the United States*. New York: Springer-Verlag, 1977.

Krimsky, Sheldon. "When Sponsored Research Fails the Admissions Test." In *Universities at Risk: How Politics, Special Interests, and Corporatization Threaten Academic Integrity*, edited by James Turk, 70-94. J. Lorimer & Co., 2008.

Levy, Steven. *Hackers: Heroes of the Computer Revolution*. Garden City, New York: Anchor Press/Doubleday, 1984.

Rosenzweig, Roy. "Wizards, Bureaucrats, Warriors, and Hackers: Writing the History of the Internet." *The American Historical Review* 103, no. 5 (1998): 1542.

Turner, Fred. *From Counterculture to Cyberculture: Stewart Brand, the Whole Earth Network, and the Rise of Digital Utopianism*. Chicago: University of Chicago Press, 2006.