Free Software and Political Economy

(Chapter 1, *Decoding Liberation: The Promises of Free and Open Source Software*, Routledge, May 2007)

By:
Samir Chopra and Scott Dexter {schopra, sdexter}@sci.brooklyn.cuny.edu
Department of Computer and Information Science
Brooklyn College of The City University of New York
2900 Bedford Avenue
Brooklyn, NY 11210

> For us, open source is capitalism and a business opportunity at its very best.
> --Jonathan Schwartz, President and Chief Operating Officer, Sun Microsystems (Galli 2005)
> The narrative of the programmer is not that of the worker who is gradually given control; it is that of the craftsperson from whom control and autonomy were taken away.
> --Steven Weber (Weber 2004, 25)

Free software, in its modern incarnation, was founded largely on an ideology of "freedom, community, and principle," with little regard for the profit motive (Stallman 1999, 70). Yet today FOSS makes headlines daily[1] as corporations relying on "open source development" demonstrate remarkable financial success (Vaughan-Nichols 2005, ; Brown 2006), those that before hewed to closed-source development and distribution now open significant parts of their code repositories (Farrell 2006) and develop partnerships with open-source firms (Haskins 2006, ; Krill 2006); and free and open-source software licensing schemes become challenges to be tackled by legal departments of software firms keen to draw upon FOSS's unique resources (Lindquist 2006).

FOSS incorporates a complex of political ideologies, economic practices, and models of governance that sustain it and uphold its production of value. It is a political economy whose cultural logic cannot be understood without reference to the history of computing and software development, the arc of which traverses some of the most vigorous technological innovation in history, one heavily reliant on the marriage of industry and science. This historical context provides a framework both for placing FOSS's salient characteristics and ideologies in dialog with theories of political economy and for examining the import of the distinction between the free and open source movements. In particular, political economy provides a useful lens through which to understand the ways in which free and open source software invoke and revise traditional notions of property and production. While the principles of the free software movement were originally orthogonal to proprietary software and its attendant capitalist culture, the recent Open Source Initiative seeks greater resonance between them. Examining the process of this convergence enables an understanding of how capitalism is able to co-exist with, and possibly co-opt, putative challenges to its dominance of the industrial and technological realms.

A Brief History of Computing and Software Development

The term “software” encompasses many modalities of conveying instruction to a computing device, marking out a continuum between programming the computer and using it, from microcode and firmware--“close to the metal”--to the mouse clicks of a gamer. Our contemporary interactions with computers, whether punching digits into cell-phones or writing books on word processors, would be unrecognizable to users of fifty years ago. Then, computers were conceived as highly specialized machines for tasks beyond human capacity; visions of widespread computer use facilitating human creativity were purest speculation. This conception was changed by successive waves of user movements that radically reconfigured notions of computing and its presence in everyday life.

The postwar history of computing is marked by steady technological advance punctuated by critical social and economic interventions. In the 1950s, IBM introduced computing to American business; in the late 1950s, at MIT, the hacker movement established the first cultural counterweight to corporate computing; in the 1970s, personal computers demonstrated a new breadth of applications, attracting a more diverse population of users; and in the 1990s, the Internet became a truly global information network, sparking intense speculation about its political, cultural, and, indeed, metaphysical implications.

The history of software is intermingled with that of hardware; innovations in software had little effect on the practice of computing unless they were in sync with innovations in hardware.[2] In particular, much of computing's early technical history concerns hardware innovations that removed bottlenecks in computational performance. Software and programming emerged as a separate concern parallel to these developments in hardware. During the Second World War, J. Presper Eckert and John W. Mauchly designed and built the Electronic Numerical Integrator and Computer (ENIAC) at the University of Pennsylvania for the purpose of calculating artillery ballistic trajectories for the US Army (Weik 1961). The first electronic general-purpose computer, the ENIAC, which became operational in 1946, would soon replace Herman Hollerith's tabulating and calculating punch-card machines, then made popular by IBM. But it was Eckert and Mauchly's next invention, the Electronic Discrete Variable Automatic Computer (EDVAC), designed with the significant collaboration of the prodigious mathematician John von Neumann, that represented true innovation (Kempf 1961). It was one of the first machines designed according to the "stored program" principle developed by von Neumann in 1945. Before stored program computing, "programming" the computer amounted to a radical reconstruction of the machine, essentially hardwiring the program instructions directly into the machine's internals. But with the stored program principle, which allowed programs to be represented and manipulated as data, humans could interact with the machine purely through software without needing to change its hardware: as the physical hardware became a platform on which any program could be run, the computer became a truly general-purpose machine.

The term "programming a computer" originated around this time, though the relatively mundane "setting up" was a more common term (Ceruzzi 2003, 20). The task of programming in its early days bore little resemblance to today's textual manipulations and drag-and-drop visual development environments. The electromechanical programmable calculator Harvard Mark I, designed by Howard Aiken, accepted instructions in the form of a row of punched holes on paper tape; Grace Murray Hopper, who joined the Mark I project in 1944, was history's first programmer. The Mark I was not a stored program computer, so repeated sequences of instructions had to be individually coded on successive tape segments. Later, Aiken would modify the Mark I's circuitry by hardwiring some of the more commonly used sequences of instructions. Based on this successful technique, Aiken's design for the Mark III incorporated a device, similar to Konrad Zuse's "Programmator," that enabled the translation of programmer commands into the numerical codes understood by the Mark III processor.

With the advent of the stored program computer, general-purpose computers became their own programmators. The first software libraries consisted of program sequences stored on tape for subsequent repeated use in user programs. This facilitated the construction, from reusable components, of special-purpose programs to solve specific problems. To some observers, this project promised to do for programming what Henry Ford had done for automobile manufacturing: to systematize production, based on components that could be used in many different programs (Mahoney 1990). Von Neumann's sorting program for the EDVAC was probably the first program for a stored program computer; Frances Holberton wrote similar sorting routines for UNIVAC data tapes.

In 1946, the University of Pennsylvania, striving to eliminate all commercial interests from the university, pressured Eckert and Mauchly to relinquish patent royalties from their inventions. The pair resigned, taking the computer industry with them, and founded the company that developed the UNIVAC. In the early days of commercial computing, software innovations were typically directed towards making usable machines for the small user community that constituted the embryonic computer market. The US Census Bureau was one of Eckert and Mauchly's first customers; Eckert and Mauchly's machines found application in inventory management, logistics, and election prediction. In an early acknowledgement that computing cultures, such as they were, were different in universities, Eckert and Mauchly's first corporate customer, General Electric, had to convince its stockholders that it had not come under the sway of "longhaired academics" (Ceruzzi 2003, 33).

In May 1952, IBM ended UNIVAC's monopoly, joining the business of making and marketing computers by introducing the IBM 701. Thus began the epic cycle, part and parcel of the computing industry's mythology, of dominant players upstaged by nimble upstarts. By 1956, IBM had shot past UNIVAC on the basis of its technical superiority and refined marketing techniques, with other corporate players such as Honeywell, GE and RCA fuelling the competition. IBM had become the dominant player in the computing sector by 1960,[3] to the extent that the Justice Department, suspecting anti-trust violations, began to show an interest in the company's business practices[4] that would culminate in the longest running anti-trust case of all time, starting in 1969. As curiosity grew about its internal governance techniques, IBM was investigated by the media as well; the magazine *Datamation*, for example, extensively covered IBM's internal affairs. Not all this media attention was uncritical: IBM was accused of being a poor technical innovator, of taking over projects brought to maturity by others, of lagging behind others in technical advances (Ceruzzi 2003). These charges would echo in the 1990s as Microsoft achieved similar dominance of the software world.

The IBM 7094, the first classic mainframe, used in scientific, technical and military applications, was announced in 1962. It came complete with lab-coated attendants, whirring tape drives, sterile white-walled data centers, and highly restricted access for both machine operators (responsible, for example, for the logistics of loading programs) and programmers. While the real control lay in the hands of the programmers, few were ever allowed in the computer room. Given the high costs of computing time, no programmer ever interacted with a machine directly. All jobs were run in batches, groups of jobs clustered together to minimize idle processor time. Programmers developed their work on decks of punched cards that were transferred to a reel of tape mounted by the operators onto a tape drive connected to the mainframe. Many programmers never saw the machines that ran their programs. The operators performed the tasks of mounting and unmounting tapes, reading status information and delivering output. Contrary to appearances, the work was not intellectually challenging: indeed, most operator tasks were later relegated to collections of software known as operating systems. In case of system malfunctions, however, operators were able to load programs by flipping switches to alter the bits of machine registers directly. Though tedious, this manipulation gave the operators an intimate relationship with the machine.

The advent of transistorized circuits brought increased power efficiency and circuit miniaturization; computers became smaller and cheaper, making their way to universities. While these machines were intended for use in data processing courses, i.e., for strictly pedagogical purposes, at university computer centers students began a pattern of "playing with computers." Users were envisioning applications for computers that went beyond their intended purposes.

By 1960, the structures of commercial and academic computing were much the same, organized around computing centers that separated programmers from the machines they programmed. The drawbacks of the batch processing model were first noticed in universities as the discipline of computer programming was first taught: batch processing greatly constrained the amount of iterative feedback available to the student programmers. Thus, the needs of the university, and the breadth of its user base, began to manifest themselves in academic requirements for computing. In a foreshadowing of contemporary university-corporate relations, IBM offered its 650 system to universities at massively discounted rates under the condition they would offer computing courses. This strategy marked the beginning of direct corporate influence on the curriculum of academic computing departments, one of the many not-so-benign relationships between university and corporation that contributed to the industrialization of the science (Noble 1979).

Programming Languages

The first programming languages, called "assembly languages," simply provided mnemonics for machine instructions, with "macros" corresponding to longer sequences of instructions; large libraries of such macros were maintained by user installations. These languages used named variables for numeric memory addresses to enhance human readability; "assembler" programs managed these variable names and oversaw memory allocation. Given their utility to programmers, it is no surprise that the first assembler for the IBM 704 was developed in collaboration with the SHARE user group (see below). Computing pioneers had not foreseen the widespread nature of the activity of programming: it developed in response to user needs and was largely driven by them.

Even early in the development of the programming discipline, the cohort of programmers was regarded as a priesthood, largely because of the apparent obscurity of the code they manipulated. This obscurity was a primary motivation for J. H. Laning and N. Zierler's development of the first "compiler," a program that automatically translated programming commands entered by users into machine code. But this compiler was significantly slower than other methods for performing such translations. The continuing desire for efficient automatic translation led to the development of high-level programming languages, which also promised to dispel some of the mystery surrounding the practice of programming.

This phase of software's history was one of heavy experimentation in programming language design. The design of FORTRAN (FORmula TRANSlation)--developed in 1957 for the IBM 704--made it possible to control a computer without direct knowledge of its internal mechanisms. COBOL (COmmon Business Oriented Language), a self-documenting language with long variable names, was designed to make programming more accessible to a wider user base. More languages were developed--ALGOL (ALGOrithmic Language) and SNOBOL (StriNg Oriented symBOlic Language) among them--each promising ease of use combined with technical power. These developments defined the trajectory of programming languages toward greater abstraction at the programmer level. While programmers still had to master the syntax of these languages, their design allowed programmers to focus more exclusively on the logic of their programs.

In 1954, a cooperative effort to build a compiler for the IBM 701 was organized by four of IBM's customers (Douglas Aircraft, North American Aviation, Ramo-Wooldridge, and The RAND Corporation) and IBM. Known as the Project for the Advancement of Coding Techniques (PACT), this group wrote a working compiler that went on to be released for the IBM 704 as well (Kim 2006). In 1955, a group of IBM 701 users located in Los Angeles, faced with the unappealing prospect of upgrading their installations to the new 704, banded together in a similar belief that sharing skills and experiences was better than going it alone. The group, called Society to Help Alleviate Redundant Effort (SHARE), grew rapidly--to sixty-two member organizations members in the first year--and developed an impressive library of routines that each member could use. The founding of SHARE--today still a vibrant association of IBM users with over twenty thousand members[5]--was a blessing for IBM, as it accelerated the acceptance of its equipment and likely helped sales of the 704. As it grew, the group developed strong opinions about IBM's technical agenda; IBM had little choice but to acknowledge SHARE's direct influence on its decisions. SHARE also contributed significant software for IBM products, ranging from libraries for scientific computing to the SHARE Operating System (SOS).

It became increasingly common for the collaborative effort among corporations and users to produce vital technical components such as operating systems, as important to system usability as high-level programming languages: the FORTRAN Monitor System for the IBM 7090 was developed by a user group, while, in 1956, another group at the GM Research Laboratories developed routines for memory handling and allocation. In 1959, Bernie Galler, Bob Graham, and Bruce Arden at the University of Michigan, in order to meet the pedagogical needs of the student population, developed the Michigan Algorithmic Decoder (MAD), a programming language used in the development of RUNOFF, the first widely used text-processing system. In the fledgling world of computing, user co-operation and sharing was necessary; thus, the utility of collaborative work in managing the complexity of a technology was established early.

Though IBM developed system software for its legendary 360 series, it was only made usable through such user efforts. So onerous were the difficulties experienced by IBM in producing the 360 that software engineer Frederick Brooks was inspired to set out a systematic approach to software design based on division of labor in the programming process. Championed first by Harlan Mills in a sequence of articles written in the late 1960s (Mills 1983), these ideas about software design were presented by Brooks in his ground-breaking 1975 text on software engineering, *The Mythical Man-Month* (Brooks 1995). Mills and Brooks, acknowledging that the software industry was engaged in a "manufacturing process" like no other, laid out principles by which the labor of creating source code may be divided among groups of programmers to facilitate the efficient development of high-quality code. This industrial move introduced division of labor to emphasize efficiency: from the beginning, industrialization was pushed onto computer science, with long-term implications for the practice of the science.[6] The complexity of software that Brooks described was recognized throughout the industry by 1968, when NATO's Science Committee convened the first conference on software engineering at Garmisch, Germany. Concurrently, pioneering computer scientist Edsger Djikstra published his influential letter, "Go-To Statement Considered Harmful" (Djikstra 1968), in an attempt to move programming to a more theoretical basis on which his new paradigm of "structured programming" could rest.

The year 1968 also saw a significant discussion of intellectual property issues take place on the pages of the *Communications of the Association of Computing Machinery* (*CACM*), the flagship journal of the primary society for computing professionals. In a policy paper published by the Rockford Research Institute, Calvin Mooers had argued for trademark protection for his TRAC language to prevent its modification by users. University of Michigan professor Bernie Galler responded in a letter to the *CACM*, arguing that that the best and most successful programming languages benefited from the input of users who could change them, noting in particular the success of SNOBOL, which he suggested had "benefited from 'meritorious extensions' by 'irrepressible young people' at universities" (Galler 1968). Mooers responded:

> The visible and recognized TRAC trademark informs this public . . . that the language or computer capability identified by this trademark adheres authentically and exactly to a carefully drawn Rockford Research standard. . . . An adequate basis for proprietary software development and marketing is urgently needed particularly in view of the doubtful capabilities of copyright, patent or "trade secret" methods when applied to software. (Mooers 1968)

While most computer science professionals acknowledged the need for some protection in order to maintain compatibility among different versions of a language, Galler's views had been borne out by the successful examples of collaborative development by the SHARE and MAD user groups. Significantly, Mooers's communiqué had noted the inapplicability of extant intellectual property law to software, which would continue to be a point of contention as the software industry grew. As it turned out, Galler's analysis was correct, and the trademarked TRAC language never became popular.

Pressure from the US Government, and IBM's competitors, soon led to the phenomenon of "unbundling," a significant step towards the commodification of software. In 1968, responding to IBM's domination of the market, Control Data Corporation filed an anti-trust suit against IBM. Anticipating a similar suit by the Department of Justice, IBM began to sell software and support services separately from its mainframes (though it preferred to lease its machines rather than sell them, maintaining nominal control over its products). IBM's Customer Information Control System (CICS) was its first software "product"; IBM's competitors were now able to sell software to customers who used IBM’s hardware.

Digital Equipment Corporation (DEC) adopted a different business model with the PDP-1 minicomputer, the first model in what would become the very successful PDP line. The name, Programmed Data Processor, was deliberately chosen to avoid some of the connotations of “computers” as unwieldy and overpriced.[7] DEC sold the PDP-1 complete with system software, and, like IBM, encouraged user modifications to the system. DEC’s policy, because it did not have the internal resources to develop software to enhance the PDP-1’s functionality and user-friendliness, was to encourage its customers to participate in the ongoing development of its products. DEC thereby explicitly acknowledged the value that customers’ knowledge added. To further support this approach, DEC adopted a *laissez-faire* attitude towards its ‘intellectual property,’ going so far as to provide copies of its technical manuals, on cheap newsprint, to its customers.

Hacker Cultures

DEC’s close relationship with one class of customer, the university, enabled it to stay at the cutting edge of innovation. Universities were attracted to the openness and flexible power of the PDP-1, which became an integral part of the hacker cultures that emerged on campuses nationwide. In 1961, MIT acquired the PDP-1, and the Signal Committee’s Tech Models and Railroad Club (TMRC) adopted it as their plaything, using it to create games, music programs, and other diversions. The complex culture that grew up around this machine rejected the corporate, high-priest-run, batch-processing style of operation that characterized IBM’s products (the TMRC coined the perjorative “coolie” to describe the industrial style writing of code (Levy 1994, 57)), valuing instead inquiry, interaction with technology, and community governance. It introduced a distinctive lexicon, social conventions, and community ethic: in short, a new and thriving subculture.

In *Hackers*, his history of computing pioneers, Steven Levy (Levy 1994, 40-45) enumerates the tenets of the new community’s “hacker ethic” as follows:

1. Access to computers--and anything which might teach you something about the way the world works--should be unlimited and total. Always yield to the Hands-On Imperative!!

2. All information should be free.
3. Mistrust authority--promote decentralization.
4. Hackers should be judged by their hacking, not bogus criteria such as age, race, class or position.
5. You can create art and beauty on a computer.
6. Computers can change your life for the better.

The hacker ethic contains the seeds of many defining characteristics of the culture of free software: resistance to the domination of technical standards and code by one central authority, the belief in a meritocracy underwritten by technical competence, the fierce protection of the freedom of information, and the belief in the social and personal transformative potential of computing. What we now think of as the "free software sensibility" is easily discerned in stories like that of TMRC hacker Stewart Nelson, who began hacking by disassembling telephones to understand their functioning. This in turn led to an understanding of how the telephone system worked, how telephones interacted with other devices; he saw no legitimate reason why these explorations should be subject to technical or socio-political constraints. The road from these investigative actions to modifying the code of an operating system so that it works correctly with a new printer is a straight one: each action pushes the limits of understanding through independent experimentation and "tinkering." Early hacker culture and the contemporary free software movement are thus part of a narrative continuum about taking control of technology and preserving user autonomy.

Several hardware generations after the PDP-1, DEC's PDP-10 became the locus of several sites of hacker culture, such as MIT's AI Lab and Project MAC, Stanford's Artificial Intelligence Laboratory, and Carnegie Mellon University. Programmers nourished by these rich hacker cultures went on to do seminal work in operating systems, artificial intelligence and theoretical computer science. Xerox's Palo Alto Research Center (PARC), arguably the single largest producer of technical innovation[8] in the post-mainframe era, also supported a thriving hacker community. Stanford and PARC would later become focal points of the entrepreneurial frenzy that transformed California's Santa Clara Valley into Silicon Valley (O'Mara 2005). The PDP-10 was also the dominant machine on the early ARPANET, a tiny assemblage of university and government computers that would become the Internet. Back at MIT, researchers had rejected the system software provided by DEC for the PDP-6 and instead developed their legendary operating system, the Incompatible Time-sharing System (ITS), which became the longest running time-sharing system[9] of all time. The language, humor, and values developed by the ITS hackers form the foundation of our contemporary understanding of hacker culture.

Another hacker's delight was just around the corner: 1969 witnessed the birth of not just ARPANET but also Unix. Designed and written by Ken Thompson and Dennis Ritchie at AT&T Bell Labs on a DEC PDP-7, it was the first operating system to be written in a machine independent medium, the high-level language called C (though it was initially implemented in assembly language). As a consequence, it was easily ported to other hardware platforms. Its popularity was largely due to its provision of a common software environment across a diverse range of hardware. The marriage of Unix and C created a new variation of hacker culture, centered on the extraordinarily rich and flexible Unix programming environment that supported intercommunicating programs and networking.

Significantly, because US Department of Justice anti-trust provisions kept communications companies out of the computing business, Unix was not sold by AT&T but licensed at no charge, with source code included.[10] The first operating system to include core Internet software like the TCP/IP networking protocols, Unix provided substantial nourishment to the communication culture, fuelled by email and bulletin boards, which had sprung up around the ARPANET and its backbone PDP-10 sites. Typically, the use of these computing facilities for personal communication was unauthorized, but funding agencies were more than willing to pay this price for fostering a brilliantly innovative community of collaboration. This networking culture culminated in the 1980s in the Usenet, the premier meeting ground and information exchange for an entire generation of Internet users (Hauben, Hauben, and Truscott 1997).

Parallel to these developments in institutional sectors, a thriving community of hobbyists, enthusiasts, and entrepreneurs was experimenting with the implications of new developments in microprocessors, in what would become known as personal computing. As they explored, well before the advent of the IBM PC, what could be accomplished with hardware of limited computational power, they believed software should be shared freely both to further innovation and to spread the word about the growing power of personal computing. These computing enthusiasts bought and used computing kits consisting of hardware components and instructions for their assembly; they wrote fledgling operating systems and utility programs; they met frequently to discuss problems, solutions, and computing experiences. Alternative economies of exchange in an environment where knowledge was a valuable commodity did not take long to emerge: the expertise of each was available to all, and remuneration simply meant having a solution provided in exchange for another (Levy 1994). In the April 12, 1975 issue (the second) of the Homebrew Computing Club's newsletter, the cover editorial enumerates some members' ideas for the club's activities and mission:

> Perhaps the club can be a central REPRO & dissemination point for hard-to-otherwise-get listings & schematics, paper tape sources and binaries AS WELL AS a place where software written in PL/M be compiled, simulated. etc., for creating working or usable binaries. . . . [M]eet to exchange ideas, share skills, problems and solutions. . . . Particularly maintain a local resource file with reciprocal arrangements with contiguous groups. . . . Exchange information. . . . mostly an information and learning center. . . . to offer a chance to get together and exchange ideas on software and hardware. . . . serve as information exchange medium; run technical discussion & education sessions. . . . share skills. . . . perform want-ad matching so that people can find what they want to have. Generally useful software, individual specific routines people have written. . . . share ideas and stop trying to have so many small business men trying to make a few dollars. (Moore 1975)

By 1975, three separate hacker cultures were thriving: the ITS community at MIT, the Unix/C networked crowd, and the personal computing enthusiasts, located largely on the West Coast. Although each group had its favorite hardware and programming languages, they jointly subscribed to the hacker ethic with its love of sharing and the belief that computing was a Good Thing. These cultures suffered blows: DEC terminated its support of the PDP-10, and the ITS system, weakened by its lack of portability, soon died out. In the early 1980s, Unix workstations, especially those designed by Sun Microsystems (founded by Unix hackers from Stanford), became wildly popular. These relied on graphical windowing systems. The most popular, the X Window System, continued to rely on the cultural standards of sharing code: its code was given away, and users fine-tuned their versions for the local environment before releasing patches back to the X Consortium.[11]

By 1984, around the time of the divestiture and breakup (enforced by the US Department of Justice) of AT&T, two communities of hackers remained: the Unix community--which had largely absorbed the ITS community--and the PC community. But Unix became a proprietary system; as each vendor added its own proprietary features, customers and vendors alike squabbled over which version would be the One True Unix. Most significantly, its flourishing culture of networked innovation on a common platform began to dissolve. As the hardware compatibility that was Unix's main strength vanished, vendors tried unsuccessfully to tap into the PC market by offering Unix-like systems for personal computers, though without the access to source code that the original Unix systems provided. Vendors' failures to understand the importance of providing source code to the creation of an independent and empowered user group meant that no hacker culture grew in the Unix/PC environment. The Unix community would continue to flounder till the early 1990s, when Linus Torvalds released his embryonic Linux kernel for personal computers.

The Continuing Commodification of Software

Mainframe vendors, and their customers, treated software as purely ancillary to the 'real' product: the hardware (IBM's unbundling, under duress, was more the exception than the rule). In the nascent communities of hackers, PC enthusiasts, and university researchers, the notion of software as a good that could be sold--or property that could be stolen--was an alien one, until Bill Gates's 1976 open letter to the Homebrew Computer Club, in which he accused its members of stealing his software. With Microsoft still in its infancy, Gates was only the co-author of a small (if important) program, a translator for the BASIC programming language, which allowed most hobbyists to experiment effectively with their Altair computers. In his letter, he notes the *laissez faire* attitude of hobbyists toward software ownership, makes several crucial claims about the economic structure of the emerging software market, and concludes by accusing the community of widespread theft:[12]

> To me, the most critical thing in the hobby market right now is the lack of good software courses, books and software itself. Without good software . . . will quality software be written for the hobby market? . . . Paul Allen and myself . . . developed Altair BASIC. The value of the computer time we have used exceeds $40,000. Two surprising things are apparent: 1) Most of these "users" never bought BASIC (less than 10% of all Altair owners have bought BASIC), and 2) The amount of royalties we have received from sales to hobbyists makes the time spent on Altair BASIC

> worth less than $2 an hour. Why is this? As the majority of hobbyists must be aware, most of you steal your software. Hardware must be paid for, but software is something to share. Who cares if the people who worked on it get paid? Is this fair? One thing you do do is prevent good software from being written. Who can afford to do professional work for nothing? What hobbyist can put 3-man years into programming, finding all bugs, documenting his product and distribute for free? The fact is, no one besides us has invested a lot of money in hobby software . . . there is very little incentive to make this software available to hobbyists. Most directly, the thing you do is theft . . . Nothing would please me more than being able to hire ten programmers and deluge the hobby market with good software. (Gates 1976)

This conflict between Gates and the Homebrew Computing Club's members was indicative of an emerging tension in the hacker community (Levy 1994). Some simply wanted to use machines for the sake of using machines (embodying the "tools-to-make-tools" philosophy), some saw computing technology as carrying political and normative implications, and some were especially sensitive to the commercial possibilities of this technology. This divergence would also underwrite the schism, in the 1990s, between the free software and open source movements.

These tensions in the hacker community were manifested most notably in the conflict among MIT, Lisp Machines, Inc. (LMI), and Symbolics, Inc., a conflict that indirectly resulted in the birth of the GNU free software project. In the late 1970s, members of MIT's Artificial Intelligence laboratory had developed a prototype computer, the "Lisp Machine," specially optimized for programs written in the Lisp programming language. Sensing commercial potential for this machine, lab members began discussing possible commercialization. After strong disagreement within the lab over business tactics, Symbolics and LMI were spun off, both staffed by former lab members and hackers. The Lisp Machine system software was copyrighted by MIT, used by Symbolics under license, and also shared with LMI. The actual source code was kept on an MIT server, where both Symbolics and MIT staff could make changes, though changes and improvements were openly available to all three parties. This system persisted until 1981, when, in a dispute with MIT, Symbolics began to insist that MIT's improvements be made available only to Symbolics, effectively stifling LMI, which was unable to maintain its own version of the software. Richard Stallman, one of the only members remaining in the AI lab, described the impact of Symbolics's decision:

> [T]hey demanded that we had to choose a side, and use either the MIT version . . . or the Symbolics version. . . . If we worked on and improved the Symbolics version, we would be supporting Symbolics alone. If we used and improved the MIT version . . . we would be doing work available to both companies, but Symbolics saw that we would be supporting LMI because we would be helping them continue to exist. . . . Up until that point, I hadn't taken the side of either company, although it made me miserable to see what had happened to our community and the software. But now, Symbolics had forced the issue. So, in an effort to help keep Lisp Machines going--I began duplicating all of the improvements Symbolics had made to the Lisp machine system. I wrote the equivalent improvements again myself. . . . [F]or two years, I prevented them from wiping out Lisp Machines Incorporated, and the two companies went on. . . . Meanwhile, it was time to start building a new community to replace the one that their actions and others had wiped out. (Stallman 2002)

Stallman had already experienced the technical frustrations of having code closed off:

> The . . . AI Lab received a graphics printer as a gift from Xerox around 1977. It was run by free software to which we added many convenient features. For example, the software would notify a user immediately on completion of a print job. Whenever the printer had trouble, such as a paper jam or running out of paper, the software would immediately notify all users who had print jobs queued. . . . Later Xerox gave the AI Lab a newer, faster printer, one of the first laser printers. It was driven by proprietary software that ran in a separate dedicated computer, so we couldn't add any of our favorite features . . . no one was informed when there was a paper jam, so the printer often went for an hour without being fixed. The system programmers at the AI Lab were capable of fixing such problems. . . . Xerox was uninterested in fixing them, and chose to prevent us, so we were forced to accept the problems. (Stallman 1992)

Stallman's response to this state of affairs was a radical plan to develop an entire system built of free software and, further, to think about how a hacker's software could remain free software. Symbolics had shown it was possible to close freely available code, to build on the efforts of others but not give back in turn. Stallman's system, dubbed GNU (for the recursive acronym "GNU's Not Unix"), while clearly pragmatic from a technical perspective, embodied a broader philosophical and social goal: to replicate and disseminate the ideals of freedom and cooperation that characterized much of hacker culture, particularly that of the golden days of the MIT AI lab.

Stallman's original newsgroup announcement of his intention to develop a system built exclusively from free software did not distinguish between "free" as in liberty and "free" as in price, though it does acknowledge the existence of a community with shared values:

> FromCSvax:pur-ee:inuxc!ixn5c!ihnp4!houxm!mhuxi!eagle!mit-vax!mit-eddie!RMS@MIT-OZ
> From: RMS%MIT-OZ@mit-eddie
> Newsgroups: net.Unix-wizards,net.usoft
> Subject: new Unix implementation
> Date: Tue, 27-Sep-83 12:35:59 EST
> Organization: MIT AI Lab, Cambridge, MA
>
> Free Unix!
> Starting this Thanksgiving I am going to write a complete Unix-compatible software system called GNU (for Gnu's Not Unix), and give it away free to everyone who can use it. . . .
> Why I Must Write GNU
> I consider that the golden rule requires that if I like a program I must share it with other people who like it. I cannot in good conscience sign a nondisclosure agreement or a software license agreement. So that I can continue to use computers without violating my principles, I have decided to put together a sufficient body of free software so that I will be able to get along without any software that is not free. (Stallman 1983)

These values, Stallman perceived, were being steadily eroded by the increasingly proprietary nature of commercial software. Central to GNU's objective was the practice, known as "copyleft," of providing source code so users could modify, enhance, and customize their software without restriction, as long as any distribution of a modified version also included the source code (Moody 2002). Stallman began implementing large parts of a Unix-like operating system, including development tools such as the *gcc* compiler and the *gdb* debugger, shell environments such as *bash*, the editor *emacs*, and a host of utilities. To provide organizational structure and ongoing support for these efforts, Stallman founded the Free Software Foundation (FSF) in 1985.

Stallman's announcement was also a call for help and mutual aid, both of which he would receive in good measure and which would serve as the model for the development of the most identifiable symbol of the free software movement: the GNU/Linux system.

The Rise of Free Software and the "Open Source" Schism

The Internet introduced a mode of information transfer that facilitated something near and dear to hackers: open and easy sharing of knowledge. Without the Internet, Stallman could neither have made his announcement public, nor have called upon, or expected, the kind of collaboration he received. Similarly, the development of the Linux kernel began, in 1991, with Linus Torvalds obtaining POSIX standards[13] documents electronically; posting his first release to an FTP site; and announcing the code, asking for help, and receiving bug fixes by email. Torvalds chose to release Linux source code under the GNU General Public License (written by Stallman), which ensured that any changes to the kernel code, if included in a subsequent distribution, would have to be released back to the Linux development community.

The Linux project shared with GNU the practice of distributing source code, though largely for the pragmatic value of having as wide a range as possible of talented programmers making improvements for incorporation in the "central" version. GNU/Linux became the latest in a long line of software systems, from the SOS, through much of DEC's system and applications software, to the ITS system at MIT, which were developed and maintained through collective effort. The Linux development process, further amplified both by the growing population of competent programmers on the Internet and by the combination of the Linux kernel with the GNU project's software, saw the quality and popularity of the system increase to the point that by the late 1990s, several corporations, such as IBM, announced support for GNU/Linux. A growing collection of powerful free software, such as Apache, BIND, and Sendmail, which powered the Internet, as well as GNU/Linux, had begun to demonstrate their superiority to proprietary commercial software.

In 1998, a group of respected free software developers, seeking a higher profile and greater corporate acceptance for the free software development model, launched the "Open Source Initiative" (OSI).[14] The initiative followed a successful campaign to persuade Netscape, on technical and business grounds, to open the source code for its popular Communicator suite. This commercially-oriented drive required a significant change in free software rhetoric and tactics, which quickly developed into a full-blown philosophical and tactical schism among free software adherents. Spearheaded by free software developers Eric Raymond and Bruce Perens, the OSI claimed the term "free software" was too confusing for business leaders (who apparently do not understand the difference between free speech and free beer[15]) and chose the term "open source" to increase the attractiveness of the free software development model to the business community.

Starting with this break, the free software community has been confronted with the question of how much to compromise its original ideals in the name of widespread acceptance and success of its products. While the OSI and the FSF share many philosophical and pragmatic positions, their views on the rights and responsibilities of users, their relative tolerance of proprietary software, and their relationship with the world of corporate software production are subtly but significantly different. As an example, the Free Software Foundation actively encourages the use of software licenses that ensure the freedom of the licensed software in perpetuity, while the Open Source Initiative takes no position on which of many FOSS licenses most effectively protects the movement's commitment to the availability of source code. These differences are a particular expression of a fundamental division among FOSS developers over the relative valuation of technical pragmatism and the idealism of open information exchange and dissemination.

The FOSS world's relationship with the corporate world has always been an uneasy one. While many FOSS licensing schemes embody a *laissez faire* attitude toward the commercialization of code, many developers still identify sufficiently with the old university-centric hacker ethic to feel uneasy about anything that smacks of corporate co-optation. This was evident in the reaction of the Unix community in 1982 when Bill Joy, a programmer working on Unix at Berkeley, moved to the fledgling Sun Microsystems, taking the Unix codebase with him. Even though this code was licensed as free software, and his action did not restrict the freedom of other programmers to work on the Unix code, his move was nonetheless perceived as traitorous by the community (Weber 2004, 181). This tension continues to be a significant determinant of the internal debate and future direction of the free and open source software communities.

The history of software development chronicles a complicated relationship among three groups often in conflict: large corporations engaged in the manufacture of valuable technical goods, scientific researchers working in university labs, and a loose coalition of technology enthusiasts. It is a history of the growth of one of the largest, most innovative, and most valuable industries on the planet, one whose products continue to reshape our political and economic realities. This history, inseparable from a larger history of computer science and technology, is one of a deep and ongoing struggle to determine how software should be made, used, and understood, and is therefore critical to our analysis of its political economy.

Political Economy and Software Production

Understanding the political economy of a system requires an inquiry into both its creation of value and its organizational principles: how does it operationalize concepts such as labor, value, property, and production? what are its regulative principles? how is it governed, and how are conflicts resolved? Thus, analyses of the political economy of FOSS seek to understand the roots of its economic and political viability. Steve Weber's seminal analysis of the FOSS community includes an examination of the processes by which the "management" or the "people" of free software enable internal resolution of technical or political conflicts (Weber 2004). This study has the sort of anthropological or ethnographic flavor also found in Eric Raymond's *The Cathedral and the Bazaar* (Raymond 2001). Others have studied the incentive schemes--such as shared socio-cultural norms, rational choice incentives, or reputation enhancement--that motivate developers to contribute their labor to projects without direct financial recompense (Gallaway and Kinnear 2004, ; Benkler 2002, ; Boston Consulting Group 2002, ; Ghosh and Glott 2002, ; Cusumano et al. 2005, ; Ghosh 1998, ; Himanen 2002, ; Weber 2004, ; Kollock 1997, ; Benkler 2006, ; Hippel and Krogh 2003, ; Lancashire 2001, ; Lerner and Tirole 2001). These studies are an unassailable argument that the free software phenomenon challenges traditional assumptions about organized intellectual and economic activity. When intangible, non-rival goods alter modes of production and make possible economic exchanges reliant not only on monetary incentive schemes, classical political economy fails.

Classical Analysis of Software Production

Viewed through the lens of Karl Marx's and Adam Smith's commodity-centric theories, the software industry is based on the production, consumption, and exchange of a good bought and sold in accordance with the value placed on it by the market. This scarcity-based analysis necessarily assumes limited amounts of software can be produced and suggests the price of the good is a function of market pressures.

Such analysis, under the further assumption of society's increasing need for information-related goods, predicts the demand for programming skills will constantly expand, with the price paid for these skills also determined by the market. This determination takes into account programmer skill, education, and market forces such as capacity, supply, and demand. The interactions between sellers and buyers determine the wages of software workers. Programmers strive to find the highest possible price for their skills in the market, while prices paid for programming capability reflect market demand as it responds to the expressed need of its consumers. With these conditions in place, Marx would predict, the software market, driven by the irresistible energy of capitalist enterprise, would expand to global scope.

In this classical politico-economic picture, Marxist analysis claims workers are deprived of the surplus value associated with a product. The capitalist owns the means of production, pays the worker a wage that undervalues her labor, and sells the goods at a profitable margin. The labor value provided by the worker is denied her. Further, because the worker sells her labor power to earn a living, and the capitalist owns the manufacturing process, the product of her labor is alien to the worker. This analysis would see in industrialized software production the same control and exploitation of labor. In such a system, over a period of time, software production will move to regions of low-cost labor, or, alternatively, low-cost labor will be imported to areas in which it is required. Consequently, the price paid for software labor power will diminish to the minimum necessary to maintain the class of programmers. These predictions are confirmed by recent practices ranging from outsourcing to software parks in India to US immigration policies that allot specialty worker visas to hire computer scientists at below-market rates.

The Creation of Value by Proprietary Software

In the nascent computing industry, when the provision and maintenance of hardware was the main cost of computing systems and customer bases were tiny, hardware vendors made their source code freely available, encouraging customers to modify it and develop their own. There was no independent market for software; it was subsidiary to the primary commodity, hardware. Thus, control of the means of production of computing systems stayed with the corporate owner. IBM's move to unbundle its software from its hardware introduced software as a new commodity, necessitating the creation of new controls of the means of production of this knowledge-based commodity.

Bill Gates's intervention with the Homebrew Computer Club was a move toward creating scarcity in the ballooning software market for personal computers, with his letter suggesting software could be a marketable commodity only if the prevalent practices of shared development and unrestrained code sharing were squashed. Ignoring what had been demonstrated in the corporate sector by SHARE and in the academic sector by MAD, Gates's letter asserts that personal computing user groups organized around such practices would be unable to produce high-quality software and would render the software economy unviable. That the scarcity of software is maintained by the questionable application of copyright and trade secret law[16] suggests that alternative political economies, ones that acknowledge the plenitude of software, are not implausible.

Gates's claims are merely the latest restatement of the "techno-economic determinist" story, a mythology which suggests economically successful technologies are selected through the interaction of three filters (Noble 1991). First, scientists and engineers, with their dedication to rationality and efficiency, methodically subject technological possibilities to objective scrutiny. Second, practical businessmen and financiers develop and deploy only the most economically viable technologies. Finally, the anonymous operation of the self-regulating market ensures only the most socially beneficial technologies survive the rigors of competition. Such a story has neither the space nor the need for the self-governing technological communities typified by the FOSS movement.

Once both hardware and software were commodified, the new software capitalists needed to, and indeed did, retain control of the means of production. For a knowledge-based commodity like software, the knowledge and skills of the worker are an integral component of the means of production. These are controlled, for example, through non-disclosure agreements--gag orders that prohibit workers from revealing details of their work to entities external to their employer both during and, in some cases, after employment (Radack 1994). Other knowledge is carried not only by the workers themselves but also by the source code they create; because source code is the key to software's inherent modifiability, restricting access to source code is a crucial component of the control of the means of production.

Control over the source code is established by a two-fold regime that places restrictions on the copying of the executable (by employing copyright law) and on availability of source code (treated as a trade secret). Thus, the means of production remain with the corporate owner of the software. The lack of knowledge of the code alienates users from the product as they are unable to modify the code, while the inability of the worker to receive full value for his skills alienates him from his knowledge, as he is not permitted to discuss it, let alone make it available to a larger set of potential employers.

Because proprietary software is written for profit, it must create a commercial advantage. But software has the serious disadvantage that competitors could replicate its functionality, which would rapidly destroy profits. Therefore, proprietary software, in order to meet its profit imperative, must build in technical barriers to competition (Levien 1998). Most software is inherently dependent on other components of the system in which it operates; often the usefulness of the software depends on how well it interoperates with its environment. Thus, one approach to creating value is to create an environment in which competitors would be unlikely to succeed. Proprietary software vendors are thus driven to monopolistic practices--like employing technical barriers to interoperability with competitors' products--as they attempt to control the environment in which their programs run. Microsoft, for instance, does not just write proprietary applications, it also produces the operating system that provides the environment for those applications. This bundling of operating system and application is an integral part of its competitive strategy. While the operating system is nominally open to rivals for Microsoft's applications, they cannot use its resources as effectively, for Microsoft "allows Microsoft applications to cheat, and call directly into the undocumented . . . system call interface to provide services that competing applications cannot" (Allison 2005). Thus control over source code, programmers' knowledge, and the details of the relationship between software and its environment all contribute to creating value in proprietary software.

‘Late Capitalism’

The distributed, asynchronous process of software production that led to GNU/Linux has many features in common with the globalized economy. These features are often understood as characteristic of a particular stage in the evolution of capitalism (Harvey 1989, ; Jameson 1991). Most histories of capitalism identify three stages in its evolution: the classical 19th century capitalism that was the concern of Smith and Marx, the early 20th century monopoly capitalism predicted by Marx, and the 21st century’s, distributed, multi-national, global or “late” capitalism, a cluster of notions related to a global change in industrial practices and concerns, including:

> the new international division of labor, a vertiginous new dynamic in international banking and the stock exchanges . . . new forms of media interrelationship . . . computers and automation, [and] the flight of production to advanced Third World areas. (Jameson 1991, xix)

Late capitalism, then, incorporates the globalization of capital, the movement from manufacturing to service based economies in the First World, the dispersion of labor due to porous borders and networked command and control, the imposed flexibility of labor (long working days, 24-hour factories), the increasing interconnectedness of world economies, the classification of knowledge as a commodity, and the displacement of the marginal cost model of production:

> [T]he informational revolution will transform industry by redefining and rejuvenating manufacturing processes. . . . Just as through the process of modernization all production tended to become industrialized, so too through the process of postmodernization all production tends toward the production of services, toward becoming informationalized. (Harvey 1989, 120)

Many programmers participate in FOSS communities in order to trade on their knowledge and skills (Lakhani and Wolf 2005, ; Ghosh and Glott 2002, ; Boston Consulting Group 2002, ; Lerner and Tirole 2001, ; Raymond 2000). FOSS programmers can support themselves by selling their expertise in programming by offering post-sale services and customizations, rather than relying, however indirectly, on sales of the products they develop. The willingness of programmers to share their code demonstrates their understanding that knowledge is the truly valuable commodity, not the ephemeral products they make. This understanding of commodity and compensation is summed up in Raymond's formula for economic success, "give away the recipe, open a restaurant."

As an international community, FOSS employs the characteristic logic of late capitalism: decentralized power, distributed sovereignty, and a reworked concept of commodity that stresses the provision of services rather than the accumulation of physical goods (Harvey 1989, 159). Its labor force, consisting of a vast pool of highly contingent labor, is dispersed across all time zones, as workers write new code or search for flaws in others' code before sending suggested modifications to a central assembly point. These projects employ powerful code management systems: while there is a flurry of fixes, what is actually admitted to the codebase is strictly controlled by social and technical constraints (Weber 2004). FOSS development, with its flexible labor force, global extent, reliance on technological advances, valuation of knowledge, and production of intangibles, has fully embraced the modern knowledge economy.

This largely monolithic treatment of FOSS is inaccurate in a number of ways. Individual developers bring a broad diversity of techniques, knowledge, and ideologies to the community, and attempts to place them into well-defined categories are destined to fail. But there is an unmistakable division, closely related to considerations of political economy, between the open source movement (specifically, the OSI) and the free software movement (specifically, the FSF). As we have seen, the free software movement was founded on a perspective of software as a social good; the open source movement exists to make the case that open source software is more technically efficient and therefore could create more value for commercial software enterprises. That is, the free software movement is committed to the freedom of software regardless of any constraints that this stance may impose on the accumulation of wealth, while the open source movement is more open to occasional compromises to the freedom of software when required by the commercial imperative. This distinction is not simply a theoretical one; it fuels vigorous debates about FOSS philosophy (Stallman 1992), licensing terms (The Free Software Foundation 2006), and the role of proprietary software in FOSS development (Laffoon 2005, ; Stallman 2000, ; Sweet 2001).

Despite these differences, the broad FOSS community continues to thrive, maintaining creativity and producing software with both technical and economic value. The question of how FOSS manages to deviate from traditional political economy, yet continue to produce value, remains one of deep interest.

FOSS and the Creation of Value

OSI founder Eric Raymond, in addition to being a veteran FOSS developer, is one of the first, and most widely read, observers of the FOSS political economy. In a series of essays written in the late 1990s, he offers analyses of the distinguishing features of the FOSS development model ("The Cathedral and the Bazaar" (Raymond 2001)), its unique treatment of property ("Homesteading the Noosphere" (Raymond 2000)), and its internal economic logic ("The Magic Cauldron" (Raymond 2004)).

In "The Magic Cauldron," Raymond asserts that FOSS produces value because it views the software industry as one based on service rather than manufacturing. For Raymond, a software good has both use-value and sale-value. The former is its value as a productivity multiplier; the latter is its value as a salable commodity. Traditional analyses of software, using a manufacturing model, make two key assumptions: that software development time is paid for by sale-value, and that the sale-value of software is directly proportional to development costs and use-value. Thus, this model assumes that software is like any other manufactured good. Raymond suggests, however, that this characterization is inaccurate. Most software, perhaps as much as ninety to ninety-five percent,[17] is not written for retail sale; this includes in-house customization, hardware-specific device drivers, and "embedded" code such as that found in cell phones, cars, and microwave ovens. Most programmers' time is spent on the maintenance of code; these development costs are not part of sale-value. Moreover, the value of software rapidly diminishes when its manufacturer no longer supports it: consumers' expectations of support have a direct impact on the software's sale-value.

Thus, Raymond argues, if most of software's total costs are in its maintenance, debugging and support, then charging a high price for sale and very little for subsequent service does not serve anyone well. Software vendors will concentrate on hasty releases of new products, to the detriment of after-sales service, and increasing customer bases mean more demands on service, which is too expensive for software vendors to provide. Such a state of affairs is conducive to the creation of a monopoly: most vendors will fail as they discontinue services, driving their customers to the few remaining competitors.

Lowering the price of a desirable good, however, and thereby increasing its sales, will increase the cost of the support infrastructure, pushing vendors towards a service-driven model, one based on a "*continuing* exchange of value between the vendor and the customer," (Raymond 2000, emphasis in original) that mitigates against monopoly.

When software becomes free or open source, however, only its sale-value is threatened. FOSS makes it difficult to capture sale-value because its licenses allow unlimited copying: the first copy may be sold, but it is hard to sell future copies. But Raymond provides two examples in which developer salaries are paid for by use-value. In Cisco's risk-spreading model, an inversion of the non-disclosure agreement model, the software that operates Cisco products is given away as a hedge against the departure of the developers--and the specialized knowledge they carry--and to ensure continued and future support. In the consortium-funded Apache Project, a form of precompetitive collaboration, member organizations willingly fund the development of the open source licensed Apache server, recognizing that a secure scalable webserver is a crucial part of the technical and economic infrastructure of the Internet, one which supports the economic viability of anyone reliant on Internet-based business. The creation and funding of the Mozilla Foundation's Firefox project is similarly motivated by the universal desire for users/consumers to have access to a high-quality browser (Baker 2005).

Proprietary alternatives to FOSS licensing are incompatible, both economically and technically, with the FOSS community's perspectives on value creation through software development. First, "no party (with the possible exception of the originator of a piece of code) [should] be in a *privileged* position to extract profits" (Raymond 2000, emphasis in original). Second, such licenses, with their restrictions on redistribution, introduce legal enforcement costs that must be passed on to customers. Finally, these restrictions on redistribution prevent further development of the software in different directions, which acts as a damper on both economic and technical innovation. Thus, despite licenses that fail to capture sale-value, FOSS continues to produce value via the "inverse commons": due to network effects, the value of the software in the commons increases as more people use it and contribute to its further development. Here, the anti-rival nature of software commodities becomes apparent: the most valuable code in the commons is that which is most heavily used.

The feature of open source software (Raymond particularly cites the Linux kernel) that most directly enables continuing value creation is not technical but sociological: it is easy to submit changes to the codebase--the only barrier to entry is intimate technical knowledge of the code. The number of contributors to a project, and hence its chance of success, is inversely proportional to the number of obstacles to active participation faced by a potential contributor.

Ultimately, it is the governance of the FOSS community that enables this continual production of value. In "Homesteading the Noosphere" (Raymond 2000), Raymond distinguishes hacker pragmatism from the "zealotry" and "anticommercialism" of the FSF, a reflection of the historical differences in motivations among hackers. These differences are bridged by the taboos of the culture, such as those against unauthorized distribution of changes to a project or removing a developer's name from project credits, which Raymond argues are essential to community governance. In this context, choosing a licensing scheme is a decision of political economy, one that not only creates value by attracting developers and users but also facilitates governance by making public a set of assumptions about acceptable behavior.

Recognizing that notions of property and ownership have limited applicability in domains where "property is infinitely reduplicable, highly malleable, and the surrounding culture has neither coercive power relationships nor material scarcity economics" (Raymond 2000), Raymond suggests that software ownership is properly understood as the exclusive right to redistribute modified versions of software: ownership happens when someone founds a project, inherits the original, or takes over an orphaned project. Understood this way, the FOSS notion of property is akin to that of the Anglo-American common-law theory of land tenure, which allows for ownership through homesteading, transfer of title and adverse possession, respectively. Raymond asserts similar systems of ownership emerge when "property has high economic or survival value and no single authority is powerful enough to force central allocation of goods" (Raymond 2000).

The FOSS economy's abundance of storage space, network bandwidth, and computing power, rather than yielding the power and wealth typically associated with physical property, gives rise instead to a gift economy like those found in cultures without "significant material-scarcity problems with survival goods. . . . In gift cultures social status is determined not by what you control but by what you give away" (Raymond 2000). In these cultures, the only available measure of competitive success is reputation; governance structures exist primarily to maximize and protect reputation incentives.

FOSS Governance

Political scientist Steven Weber, in *The Success of Open Source*, argues FOSS creates value through a distinctive model for the facilitation and distribution of innovation. As illustrated by the FOSS development process, this model rests on four principles: "Empower people to experiment," "Enable bits of information to find each other," "Structure information so it can recombine with other pieces of information," and "Create a governance system that sustains this process" (Weber 2004, 234). Effective FOSS governance, "setting parameters for voluntary relationships among autonomous parties" (Weber 2004, 172), solves problems of coordination and incentivization to support the kinds of innovation that create value.

The most dramatic manifestation of the failure of coordination in a FOSS project is the phenomenon known as forking. Under most circumstances, developers agree the most effective way to exploit the openness of source is to work in coordinated fashion toward some shared goal of functionality. Any of them is free, however, to break away with a copy of the code and start developing the software toward another set of goals; this may be motivated by disagreements about, for example, licensing terms, the technical direction of the project, or the effectiveness of project leadership. The software's production tree splits--"forks"--at this point; the original development proceeds along one branch, the breakaway programmer's version develops along another. Most developers see the right to fork both as one fundamental to the FOSS enterprise and as one that should be exercised exceptionally rarely as it consumes many resources and can generate gross inefficiencies. Given the tendency of FOSS developers to disagree frequently and vigorously, forking is a surprisingly rare occurrence. But the rarity of forking is also attributable to effective governance, particularly "individual incentives, cultural norms, and leadership practices" (Weber 2004, 159).

These individual incentives apply to the FOSS developer as a rational actor in a marketplace of innovation: traders who hope to offer arbitrarily many copies of their own innovation in return for individual copies of many others' innovations rely on an interoperability among items for trade, which would be diminished by forking just as the proliferation of proprietary versions of Unix diminished the market value of a Unix programmer's skills. Similarly, a developer's concern for the long-term health of his reputation mitigates against many potential forks: while initiating a fork may bring a short-term gain in authority and reputation, setting a precedent of forking may make the new branch, and the developer's reputation, similarly vulnerable. Moreover, a new fork would need to attract "followers" to enable any significant development; a developer who does not already have a substantial reputation would have difficulty successfully forking an established project.

Cultural norms inform the governance of which code is actually included in a FOSS project's codebase. The fundamental question here is one of who "owns" the project (Raymond 2000); subordinate to this is the question of precisely how ownership, or authority in general, relates to the process by which code submissions are accepted or rejected. In small projects, these decisions are one of the many responsibilities of the individual owner; larger projects such as Linux, Apache, and mozilla.org exemplify a variety of collective decision-making strategies and techniques. Orthogonal to the structural considerations of how decisions are made is "technical rationality," a norm that values dispute resolution based on "engineering culture . . . bottom up, pragmatic, and grounded heavily in experience rather than theory" (Weber 2004, 164). This mitigates against disagreements manifesting on a personal level: projects are held together by the mutual understanding that disagreements reveal intellectual and technical differences, not personal ones.

The leaders of successful large projects understand the importance of arguments based on technical rationality; they are committed to "documenting the reasons for design choices and code changes in the language of technical rationality that is the currency for this community" (Weber 2004, 167). Failures of leadership most often arise from abandoning this currency: failing to support even those contributors whose work is rejected while allowing "ego-based conflict" to persist, or making technical decisions without incorporating feedback from other developers. Linus Torvalds's success as the leader of Linux, exemplifying the best aspects of technical rationality and minimizing personal disagreement, relies in part on his ability to communicate the joy and satisfaction of programming.

FOSS governance relies on a mixture of social and technical strategies. As Torvalds learned as the size of the Linux kernel and its development group grew rapidly, "engineering principles are important because they reduce organizational demands on the social and political structure for managing people" (Weber 2004, 173). This growth presented several technical and governance challenges. As the number of new developers grew, bringing new and diverse ideas for the direction of the kernel's development, Torvalds's original design of the kernel as one large interrelated piece of code became unmanageable. The kernel was therefore redesigned as a cluster of smaller, more independent, modules of code. A similar crisis, demonstrating the challenge of forking and the role of technical solutions in solving governance problems, arose in 1998, when Torvalds began to ignore important contributions to the Linux kernel. At that time, Linux kernel developer Dave Miller maintained a "mirror" site called VGER, at Rutgers University, to make access to Linux kernel source more convenient for North American developers. Miller started accepting contributions to the kernel that Torvalds did not, which rapidly led to a *de facto* fork in the kernel's development. This fork rapidly became a rift in the entire community, as Torvalds lashed out against Miller's actions even as developers increasingly complained about Torvalds's unresponsiveness. The healing of this rift required a twofold solution: an organizational solution, in which leading Linux developers, with Torvalds, agreed on a new hierarchical scheme for handling incoming code, and a technical solution, in which veteran developer Larry McVoy offered to create new source management software, specifically designed for use in open source projects, which would greatly ease the demands on Torvalds and his lieutenants.

Governance, Labor and Co-optation

FOSS development projects, organized under a multiplicity of governance schemes, may differ in the kinds of leadership qualities they value, their shared cultural norms, project teleology, the structure of incentives, and decision-making processes. The *BSD groups[18] have a common cultural norm of valuing technological problem-solving as much as, or more than, profit or the promulgation of political ideologies (Coleman 2005). Others, such as the FSF, have broad socio-technological goals in mind.[19] The Debian group's developers have "cobbled a hybrid organizational structure that integrates three different modes of governance--democratic majoritarian rule, a guild-like meritocracy, and an ad-hoc process of rough consensus" (Coleman 2005). These glimpses of governance schemes, in the context of political economy, expose to us an eclectic mix of labor organization and management structures.

The FOSS practice of distributing source code and accreting changes submitted by user-programmers is a form of production where the traditional practice of dividing labor among a pool of programmers is enhanced, expanded, and rendered radically flexible. The labor pool for a FOSS project is not limited to a small group of workers, but is expanded, exploiting the Internet, so that the cycle of distribution and accumulation of modifications is orders of magnitude more efficient and effective than the code-sharing of the past.

FOSS, that is, relies on "distribution of labor", an enhanced form of division of labor, by opening the gates of the virtual factory. The organization of this "factory" can be hierarchical, but this governance does not imply a hierarchical imposition of work assignments. The development of the Linux kernel is controlled by one man, Linus Torvalds, sitting astride its mountain of code. He and his lieutenants maintain control over the product through a rigorous system of quality control[20] while leaving individual developers free to pursue their chosen tasks within the kernel project. To make an industrial analogy, while senior developers often act somewhat like foremen, workers are not told where to go or what to do. Instead, the shop floor is scattered with tools and partially complete work, with copious explicit instructions and advice: anyone can pick up an unassigned programming task, read the documentation, and consult with other workers through email, newsgroups, and chat. Work schedules are very weakly constrained, though there is no guarantee that a worker's contribution will ultimately be accepted: contributions to work within this "factory" come from all time zones and are accepted on the basis of a meritocratic process. Workers are free to leave with a copy of the product and to open up another manufacturing shop: programmers are always free to fork a development tree. Users are also ‘workers,’ as they may become producers of future versions of the code. Workers’ inspection of each other’s work, combined with effective management, ensures that the energies of an army of programmers, of whatever size, can effectively be focused on solving the problems of creating excellent code (Raymond 2001).

The corporatization of FOSS suggests a need, to the capitalist owner, for disciplining labor. But this disciplining is made difficult by the availability to workers of empowering technological advances, their education, and their independence as they choose schedules and work assignments. The corporate owner therefore must deploy a delicate mix of cooperation and co-optation. One form this co-optation takes is the introduction and promulgation of corporate ideals. As the FOSS community makes concessions--in the choice of licensing scheme,[21] in the mixing of proprietary and free software,[22] or in the diversion of labor to closed source projects[23]--these ideals are reinforced. The most seductive promises made by the corporate world remain those of venture capital funding, commercial success, and unlimited accumulation of wealth, all of which require such concessions. As the FOSS community negotiates with the corporate world for acceptance, as open source projects turn into commercial enterprises (Pepperdine 2004), it becomes increasingly difficult for the community to maintain the purity of the original hacker ideals. The discussions this provokes in the FOSS community bring to the fore the rift between the free software and open source movements.

The exploitation of “free labor” is another potential locus of co-optation. This “intense collective labor of programmers, designers, and workers” has played an integral role in, and placed its stamp on, the development and enrichment of the Internet:

> Simultaneously voluntarily given and unwaged, enjoyed and exploited, free labor on the Net includes . . . building Web sites, modifying software packages, reading and participating in mailing lists . . . it is the spectacle of that labor changing its product that keeps the users coming back. The commodity . . . is only as good as the labor that goes into it. . . . The notion of users’ labor maintains an ideological and material centrality. . . . [T]he best way to stay visible and thriving on the Web is to turn your site into a space that is . . . built by its users. Users keep a site alive through their labor. . . . Such a feature seems endemic to the Internet in ways that can be worked on by commercialization, but not substantially altered. (Terranova 2000)

This phenomenon clearly represents an “extraction of value” from continuous work, though this critique only points to the existence of labor uncompensated by traditional means such as money exchange: it trades on an old confusion by conflating “financially uncompensated” with “free.” Compensations in the FOSS economy are diverse; a programmer who releases code freely may do so anticipating others will release code useful for him.

But the open source movement does provide evidence of the trend toward the cooptation of the digital economy by the corporate world.[24] This cooptation is inevitable, precisely because the cybereconomy reflects the economy in which it is embedded:

> [I]t is . . . impossible to separate . . . the digital economy of the Net from the larger network economy of late capitalism. . . . [T]he Internet is always and simultaneously a gift economy *and* an advanced capitalist economy. (Terranova 2000)

The co-optation is either implicit (“you must go open-source to get access to . . . the near-instantaneous bug-fixes, the distributed intellectual resources of the Net, the increasingly large open-source code base” (Leonard 1999)) or explicit (“what better way to shed staff than by . . . having code-dabbling hobbyists fix and further develop your product?” (Horvath 1998)), and signals a lack of reciprocity, especially of code-borrowing, prefiguring the eventual depletion of the FOSS sphere:

> [T]he open source question demonstrates the overreliance of the digital economy as such on free labor. . . . [It] is part of larger mechanisms of capitalist extraction of value which are fundamental to late capitalism. . . . Late capitalism . . . exhausts [free labor] by subtracting selectively but widely the means through which that labor can reproduce itself . . . [it] . . . nurtures, exploits, and exhausts its labor force. (Terranova 2000)

The corporate tendency to extract maximum value from any arrangement of labor and production exploits a tension in this sphere between the pragmatic techno-optimists (“open-source movement”) and the quasi-altruistic communitarians (“free software zealots”). The question for the FOSS community is whether this extraction of value is to be resisted. If resistance is in order, what form might it take? As we will see, the array of FOSS licenses represents a variety of answers to these questions.

Co-optation, properly understood as the furtherance of another’s goals under the mistaken impression of furthering one’s own, could take several forms in the FOSS realm. One is the compromising of the goals of FOSS: for people to have agency over the software on their machines, and, subordinately, to write free software and encourage its widespread adoption. This compromise could be effected in a variety of ways: the number of FOSS programmers could decline, some lured away by enticing proprietary software jobs, some unconvinced by the inadequate marketing of the technical and economic viability of FOSS, some thwarted by unforeseen shifts in the economy that render FOSS a financially unviable way to work. Furthermore, FOSS products could be driven out of the market by monopolistic pressures: Microsoft's Internet Explorer, which was originally based on the open-source Spyglass browser, essentially drove Netscape out of the browser market by being bundled with the Windows operating system. Thus, continued non-reciprocal borrowing of free software for use in proprietary products could increase the latter's market share without contributing to the software commons. Lastly, and most subtly, there could be a weakening, after exposure to differing values, of the shared norms that bind the community.

The FOSS community does not have control over the first method (though sufficient exposure to FOSS norms could weaken the allure of enticing proprietary software jobs). The second could be affected by executive decisions over the deployment of licensing schemes; keeping code free, and ensuring reciprocity in code sharing through licensing provisions such as those of the GPL ensures protection against co-optation. The third is controlled by discourse in the community. Here, significantly, the rhetorical thrust of the open source community (Berry 2004), in sharp opposition to the language of the free software community, is an invitation to co-optation. The Open Source Initiative's pronouncements indicate that this language is deliberately employed in the cause of corporate acceptance. The constant refrain that there is "nothing special" about free software other than its development model-- e.g. "[Open source] is one tool among many that can . . . create business value. . . . [It] is just a way to put product in many users' hands inexpensively" (Olson 2006)--and the lack of attention paid to the cultural norms of the FOSS community are indicators of this trend. The OSI's discourse is shot through with corporate jargon; the only time the corporate world pays attention to the GPL is when legal departments warn of potential violation of its terms in the mixing of free software with proprietary packages. The OSI thus enables a subtle exploitation of the programmer whose code may well be used to further corporate ends and undermine his own community.

FOSS reworks not only our understanding of labor relations but also that of the worker's relationship to the goods he produces. Traditional arrangements of commodity production locate the knowledge and decisions of technique outside of the worker. In FOSS, the conception of the open source user as empowered programmer[25] enables a reconfiguration of the relationship between worker and product that substantially addresses Marxian notions of worker alienation. The user of proprietary software is alienated from the product; he is unable to perceive its product's infrastructure or adapt it to meet his needs. The FOSS model, by making source code available, modulates this alienation by casting users as workers who might modify the product. This blurring of the programmer/user distinction thus acts to erase the consumer/producer distinction as well. Alienation from products is mitigated in the FOSS world because the worker can derive independent profit and surplus value from his work. The knowledge-based nature of software, revealed by its source code, resolves the problem of alienation because "the worker... achieves fulfillment through work [by finding] in her brain her own, unalienated means of production" (Terranova 2000).

The potential for such fulfillment is partially embodied in programmers' right to fork. While the technical rationale for the right to fork is protection against the incompetence of one set of code maintainers, it also preserves a spirit of entrepreneurship: a FOSS worker can seek independent commercial advantage from her copy of the code. A worker could leave the "factory" one day with a copy of the software and commence sales of the software, or a derivative of it, at any price he (and the market) sees fit. The success of this breakaway project is only limited by the worker's technical competence and ability to recruit co-developers.

Furthermore, in FOSS, the extraction of surplus value from the product is under the control of the worker. To redress the absurdity of assigning a commodity value to software, a good whose marginal cost of production is zero, the market assigns it a phantom price, one that supposedly captures the cost of the labor and expertise of those who wrote it. As the sale-value of software is driven down by the marginal cost of reproduction, free software workers may leverage their freedom by selling, at market rates, services such as support and customization for products they know intimately.

Viewed through the lens of political economy, Bill Gates's intervention with the

Homebrew Computer Club, based on the reasoning that users would pay a market-determined price for a commodity they were unable to develop on their own, showed a software entrepreneur's keen understanding of the true scarcity that lay ahead. In the software world, historically, the scarcity had not been material but one of knowledge. Thus, while the problems faced by the personal computing community in the 1980s, the lack of adequate software, were the same as those faced by consumers in the 1960s, one key difference between the two groups was that by the 1980s, users had become excluded from participation in the development process. Ironically, commercial personal computing brought widespread access to computer software and hardware, but at the cost of a fundamental alienation of even the interested user from the product. The growth of the computer industry in the wake of the personal computer revolution has maintained this industrial image of the user separated from the code that runs his machines.

FOSS and Property

Like any digital good, software can be distributed, via the Internet, to facilitate a direct and "just-in-time" connection between producer and consumer. Unique to free software, however, is its close identification of its mode of production with its mode of distribution. Linux, as an archetypal example, was produced through a process of transnational accretion of repairs and improvements. This accretion was only possible because of the pre-existing distribution mechanisms of the Internet; more importantly, consumers of Linux rely on the same distribution network as its producers. This leads to a unique situation:

> [I]n the world of zero-marginal cost, anarchist distribution--that is, distribution without exclusion from the act of distributing--produces inherently superior distribution . . . when the right to distribute goods with zero marginal cost has to be bought and sold, there are inefficiencies introduced in the social network of distribution. When no such buying and selling, no such exclusion from the power of distribution exists, distribution occurs at the native speed of the social network itself. (Moglen 2003)

Adam Smith's original analysis of market economies relied on the context of particular kinds of goods, modes of production, and arrangements for exchange and compensation. But when the character of goods and production changes, the market in which these goods are exchanged must adapt on pain of gross inefficiency (Moglen 2003). The maintenance of these inefficiencies by the exercise of monopolistic power and regressive deployment of intellectual property rights is not only anathema to a free market but also certain to fail. Thus, while the “invisible hand” pushes the market toward change, these interventions resist its benign influence.

In typical markets, producers provide property, in the form of goods, to consumers, a transaction underwritten by a particular monetary scheme through which exchange value is communicated and realized. When markets are structured around goods with a radically different nature, notions of monetary compensation and property change in turn. Most observers of the FOSS world agree that economic incentives and compensation are not exclusively monetary, supplemented by reputationomics, gift cultures, network effects, barter economies and the like. But perhaps the most significant feature of the FOSS economy is its explicit disdain for, and inversion of, traditional assumptions about the importance of private property.

David Hume famously suggested that property relations only make sense under conditions of scarcity (Hume 1978, 484-498). But in the software world, the scarcity is not of the good, but of an intangible commodity, the programmer’s knowledge. That treating software as private property crucially affects its creation, use, and further development has long been recognized in the free software community: “if a program has an owner, this very much affects what it is, and what you can do with a copy if you buy one. The difference is not just a matter of money. The system of owners of software encourages software owners to produce something--but not what society really needs” (Stallman 1994). Control of the means of production of, and exclusion from, property through non-disclosure agreements, copyright, patent, and trade secret law is the fundamental scarcity-creation principle in proprietary software’s property regime.

The rejection of software ownership by the free software community denies both these principles:

> The advance of digital society, whose involuntary promoter is the bourgeoisie, replaces the isolation of the creators, due to competition, by their revolutionary combination, due to association. Creators of knowledge, technology, and culture discover that they no longer require the structure of production based on ownership and the structure of distribution based on coercion of payment. Association, and its anarchist model of propertyless production, makes possible the creation of free

> software, through which creators gain control of the technology of further production. (Moglen 2003)

The FOSS world can therefore be considered as championing a great experiment in alternative property regimes:

> [W]e have made a social network committed to the proposition that the central executable elements of human technology can be produced by sharing--without exclusionary property relations. And that if the central executable elements of technology can be *made* by sharing, without exclusionary property relations, then the non-executable elements of culture--art, useful information, and so on--can be *distributed* without exclusionary property relations. (Moglen 2003)

That is, software can be successfully conceived as common property rather than private property, with implications for the development and dissemination not only of technology but also of culture. But the artificially asymmetric accumulation of intellectual resources created by the contemporary software industry's exclusionary property relations threatens any such progress.

Immanuel Kant's theory of private property highlights a constitutional incoherence in proprietary software. Kant assumes that the use of useful objects is a good and, further, that objects are not usable unless they are owned. Kant then argues for a connection between property and agency, as it would be an insult to agency and human personality if no system were devised for the use of useful objects: "it is a duty of right to act towards others so that what is external (usable) could also become someone's" (Kant 1991). He concludes that we are obliged to act in ways that make useful objects belong to someone. But as such an arrangement relies on a collectively agreed-upon property regime within which questions of ownership can be resolved, and because the appropriation of a resource as private property imposes new duties on a collective, a property regime cannot acquire legitimacy through unilateral action: it must be ratified by all involved in such a way that everyone's interests are protected equally. If a private property regime is to be established, the question of ownership cannot be begged but must be settled on a just basis. Such an agreement has not taken place in the software world: the free software community's resistance to the proprietary regime indicates it is not a universally accepted model. Both the technical and economic feasibility of the FOSS property model suggest an alternative is at hand. The proprietary software property model, then, is incoherent: it is a system of ownership devised without a meaningful conception of property to underwrite it (Boyle 1996). The freedom to own proprietary software masks a property regime that rests on exclusion; it is a system that takes away liberties (Stallman 1992, 1994).

As a social construction, property is always subject to analysis in light of changed economic conditions or artifacts with novel characteristics. Thus property, both as a concept and as a mechanism for the distribution of resources, is always open to reform and revision (Gallie 1956). If all property systems are engaged in some distributive calculus of freedom, then no property system unequivocally contributes to liberty. Talk about property is inseparable from talk about justice: every ethical or economic justification for private property rests on the assumption that it will result in a more just distribution of resources. Locke, for instance, suggested a Pareto-optimal justification for private property: everyone will be better off in such a regime, or, at least, it will benefit some and leave others unharmed. Private property rights are grounded in a clear-cut utilitarianism: they are intended to bring about a greater social good than the alternatives. When this utilitarian calculus fails, other property regimes may become more attractive, as in the nationalization of certain tracts of parkland in the US, or the Quebecois interest in nationalizing Canada's oil resources in the wake of increased fuel prices (The Canadian Press 2005).

The story of FOSS suggests an alternative calculus that may provide a more effective route to the production and distribution of high-quality software. Software, once freed, is not just nonrival, it is antirival: the more widely the software is distributed, the greater its value becomes as the result of collective debugging and improvement. The inhibitions on software distribution imposed by exclusionary property relations thus prevent the creation of maximal value in an economy based on nonrival and antirival goods. FOSS suggests a political economy of software based on non-exclusionary, equitable distribution of its resources. Stewardship and guardianship are viable alternatives to exclusion (Weber 2004, 228). Stewards of "private" property may be restricted in their use of it to the extent that each decision regarding its use requires collective consultation. To understand a programmer as the steward of his code would be to posit a different relationship of technology to society. Software becomes a carrier of socially constructed knowledge stewarded by the programmer.

The notion of property is still crucial to FOSS: software must have an owner/licensor who can license rights to the FOSS community. But this property is effectively common, not private: its usage is determined by rules that guarantee its access to all; its use is only restricted in order to protect this universal access. Any commons, though, is susceptible to the "tragedy of the commons," in which collective ownership depresses individual incentives for upkeep of the commons, resulting in its eventual degradation (Raymond 2000). But FOSS would only be vulnerable to this devolution if a licensing scheme were to allow it: thus, the choice of license is particularly important. Here again, the ideological differences between the free software movement and the Open Source Initiative are crucial, for the OSI's greater tolerance of non-copyleft licensing schemes, which allow non-reciprocal borrowing from the software commons, could lead to a depletion of the commons.

The unique perspectives on property characterizing the FOSS movement are most clearly articulated in software licenses; some differences among FOSS licenses are best understood as varying degrees of tolerance for the privatization of goods previously held as common property. FOSS licenses aim to create a common property system in which resource usage is regulated by rules whose *raison d'etre* is to make those resources available for use by all members of society. Software licensed under the GNU Public License (GPL)--the paradigmatic free software license--must remain under this license in perpetuity: if a programmer were to make changes to software licensed under the GPL, then distribute the changed code, it must also be licensed under the GPL. This ensures that the software, as common property, remains in common. The GPL influences the commercial potential of free software: if someone were selling a GPL-licensed product and a new version were released by an independent programmer, the "original owner" could integrate the new code into the old codebase and continue selling the product as before. Because the original owner in such a scenario commands market share, typically by virtue of so-called value-added services (such as support, documentation and antecedent user community), the owner may be in a position to elbow out the new software product developed by the developer.

Alternatively, under the terms of licenses such as the Berkeley Standard Distribution (BSD) license, an independent programmer's code could be used by anyone else, who could make modifications and then release the code as proprietary. The original owner would not have access to the proprietary code, and thus would be unable to make changes or integrate it into older versions without recompense to the developer. The original unmodified code would still be open; the modifications would be the only part kept secret. Such secrets, however, are not trivial, and if juxtaposed with the superior marketing and market presence of a corporate heavyweight, could render the common version worthless in the face of a more heavily developed proprietary version.

The differences among free software licenses and their contrast with proprietary licenses hearken back to the original cleavage in the software world: is software a commodity or a common good? The software market became viable by privatizing a class of property previously understood as held in common. It also created the possibility for programmers to be perceived as highly paid technocrats, a phenomenon which took on extreme proportions after the tremendous commercial potential of the Internet became apparent. Bill Gates's claims in the Homebrew letter still remain the core of any economic argument against free software. Indeed, it is not a stretch to say they are the only economic argument against FOSS.

The Future of FOSS

We only have hints about the shape the software economy would take in the absence of exclusionary property relations: without monopolies propped up by intellectual property law, a more fragmented market could result, accompanied by diminishing salaries and a shrinking demand for programmers. But perhaps the ubiquity of computing and the new availability of code would provoke a demand for customized code and the programmers to create it. Software could then revert to its original status as object of craft, with the relationship of craftsman to object, and customer to craftsman, worked out autonomously. This would create a decentralized cadre of freely operating workers who would have eliminated the potential for monopolistic or capitalist co-optation through their particular deployment of intellectual property regimes.

This vision of a return to software's roots is opposed by perspectives on the early era of computing which hold it was not conducive to the production of quality software, as it provided very little economic competition (Glass 2005). But this argument is reductive in the extreme: it suggests only one reason for the production of quality software, and disregards a constellation of individual and social motivations. Furthermore, this analysis fails to acknowledge the limited market that could not accommodate the production of a diverse range of software products; the first user-developed software was a response to the lack of software vendors in the fledgling market of the time.

The free software political economy is not cleanly separable from the larger political economy in which it is situated; neither is impervious to the influence of the other. The changes promised by the FOSS economy will remain only partially realized so long as software, a good that sits uncomfortably in any taxonomy of goods and products, is misconstrued as a commodity, and so long as the fundamental source of alienation, the failure to recognize a programmer's knowledge as the true source of value, remains in place.

If political economy is understood as the interplay between the political forces of the 'outside' world and an 'internal' economy, then the fracturing of the free software world by the creation of the Open Source Initiative was a crucial event, as external pressures forced a splintering of the free software political economy. The OSI intends to supply merely a software engineering methodology--release code quickly, release often, involve users early in development--that can be transplanted into a corporate setting without much difficulty. IBM's adoption of open source has not caused any easing of their aggressive patenting regime; whatever the OSI's message about the openness of information, it has not permeated IBM's approach to its intellectual property portfolio.

The open source movement is the apotheosis of hackerist pragmatism. The OSI claims, in a complete rejection of the social constructionist view of technology, that a development model's viability, desirability, and success depends only on the quality of the software it produces: the best technology "just bubbles up." This denies the role of contingency in technology and ascribes an autonomy to technological evolution that is philosophically implausible and historically ill-grounded. The OSI enables a subtle co-optation of the free software model; what makes its stance attractive to corporate interests is both the possibility of drawing upon the technical skills of a motivated worker force and the "freedom" to convert to closed source in the future.

The free software movement typifies technocratic idealism, yet it resists the technocratic impulse of the corporate world because it is willing to sacrifice technical and economic goods to preserve the intangible value of the freedom of software (Stallman 1992, 1994). Arguments for open source relying exclusively on engineering and economic factors, such as quality, reliability, cost, and choice (Raymond 2000) are a considerable distance from the altruistic notion of sharing that underwrites free software ideals.

The OSI, while portraying itself as an advocate for a revolutionary model for the software economy, remains hostage to its tolerance of proprietary software. The original motivation for the Open Source Definition (and the Open Source Initiative's attendant movement away from Free Software) was the concern that the word "free" was misleading and unattractive to potential corporate supporters. But as Richard Stallman's speech/beer refrain shows, there simply is no confusion when we say "free speech"--no one imagines we are giving away speech for free. Stallman's choice of terms reflects the commodification of software and makes clear the basis of the movement to (re)claim software as a general public good. The distance that open source advocates seek from the free software movement as they seduce corporate interests reflects the success of a long campaign waged by those very interests to privatize the historically public resource of software.

---

[1] The initial commercialization of "open source" brought similar commercial hype and success: (Press March 1, 1999, ; Reuters 1999, ; Richtel 1999).

2 Our survey of the history of software relies heavily on Paul Ceruzzi's excellent *A History of Modern Computing* (Ceruzzi 2003) and Steven Levy's *Hackers* (Levy 1994), as well as Eric Raymond's treatment in "A Brief History of Hackerdom" (Raymond 2004).

[3] Thus the nickname "Snow White and the Seven Dwarfs" for the computer industry of those days: IBM plus Burroughs, UNIVAC, NCR, Control Data Corporation, Honeywell, General Electric and Xerox.

[4] IBM had already been sued in 1952 for anti-trust violations in the punched card accounting machine industry. The case was settled in 1956. See http://www.research.ibm.com/knowsoc/stories_IBMHistory.html.

[5] See http://www.share.org.

[6] See Chapter 4.

7 See http://en.wikipedia.org/wiki/Programmed_Data_Processor.

[8] Its innovations include the mouse, computer generated color graphics, a graphical user interface features windows and icons, What You See Is What You Get (WYSIWYG) text editors, the laser printer, Ethernet, and object-oriented programming.

[9] Time-sharing is a technique allowing a single computer simultaneously to service multiple users by taking advantage of the frequent lulls in interactive users' demands on computational resources.

[10] The source code of the kernel was not included, as it was written in machine-dependent assembly language.

[11] Though, as we will see, the structure of this relationship would become a source of contention.

[12] Ironically, much of the computer time Gates refers to in his letter was 'stolen' from Harvard, which prohibited the use of academic computing facilities for business purposes.

[13] POSIX is the collection of standards that describes the services that Unix operating systems make available to applications software; ideally, every Unix implementation provides these services in the same way.

[14] http://www.opensource.org.

[15] To be fair to business leaders, neither, apparently, do professors at MIT's Sloan School of Management. Witness, for instance, (Cusumano 2004) and the exchange with Richard Stallman in (Cusumano 2005) where the confusion persists.

[16] See Chapter 4.

[17] The OSI currently suggests that this figure is closer to 75%: http://www.opensource.org/advocacy/jobs.html.

[18] The * denotes the many different flavors of the BSD operating system such as FreeBSD, NetBSD, and OpenBSD, each with its own development community and attendant discourse.

[19] The FSF maintains a list of politically-inflected campaigns related to its mission; see http://www.fsf.org/campaigns.

[20] Though the possibility of rebellion against this control, through forking, is an integral part of open source development.

[21] See http://slashdot.org/articles/99/06/23/1313224.shtml.

[22] See http://en.wikipedia.org/wiki/Tivoization.

[23] Such as Linus Torvalds accepting employment at Transmeta before taking up his position at Open Source Development Labs.

[24] Worries about the excessive commercialization of the Internet are common features of speculation about its future (Boyle 2000, ; Harmon 2000, ; Lessig 2000, ; Mirapaul 1999, ; Napoli 1999). The 2006 controversy over “Net Neutrality” shows that these concerns have not abated.

[25] There is some resonance here with the notion, from Japanese management theory, of quality circles, volunteer worker groups who are given access to management to present ideas about workplace improvement.