Introduction

1.1

1984 Richard Stallmann started the GNU project. He felt that the knowledge that constitute a software should be free otherwise, few companies may dominate the industry. GPL license states that you can't charge for that piece of software and every derivative work must also comply with GPL.

1.2

Open source license permits greater liberties than GPL, in particular it allows mixing of proprietary and open-source software.

1.3

Academic and industry research have different goals, more and more innovation are industry-driven (medical domain and computer science). This mean that technology advance is mostly focused ont he industry needs.

1.4

Students that grew up during the Opens source revolution are now in the industry. Many industries offer free software and paid support.(RedHat, SendMail). Open source as a business opportunity.

1.5

Computer science follows the scientific method, so it must be replicable. This means that source code must be shared (otherwise there is no way to replicate an experiment). By sharing the code open source developers make it more robust as more people can spot bugs.

1.6

Most software ventures, like scientific enterprises, have a high failure rate. Open-source and proprietary software face similar challenges in achieving success. Innovation in both fields comes with a cost, as maintaining control over an active open-source project can be challenging. The fear of losing control hinders some individuals and companies from participating. Concerns arise about code forks, where a code base diverges into separate, incompatible paths. Linux has managed to avoid major forks due to the open methods used in its development. While Linux has limited forking, there are patches that create specialized versions for specific applications without fracturing the Linux community. Similar to scientific theories, different software ventures can conflict, and the history of Lucid, a failed company that developed Xemacs, serves as an example. However, even failed enterprises contribute to the longevity of open-source software, as long as there is a demand for it. Xemacs, despite having fewer developers, remains a popular and evolving product.

1.7

Programmers often have deep loyalty to projects beyond compensation. Writing and giving away free software isn't purely altruistic. It goes beyond work and conventional notions of reward. Programmers code out of passion, finding satisfaction and intellectual stimulation. Sharing and collaboration are vital, as reputation and value come from sharing work. Empowerment and addressing frustrations drive open-source projects. Free software is a gift culture, emphasizing reputation and the importance of sharing. It's a new economic model and culture, where coding is seen as a defining aspect of intellect.

1.8

Hackers, motivated by intellectual pursuits, are now finding pragmatic and opportunistic avenues in the realm of Open Source. In Silicon Valley, where investment and venture capital drive the economy, the focus is shifting towards Linux and Open Source-related companies. The decline of the Internet IPO wave has paved the way for increased scrutiny and a demand for profitability. The question is not whether venture capital will flow towards Open Source, but why it took so long to gain traction. Microsoft's dominance in the commercial market has created a creatively oppressive environment, leading to the rise of free software as an alternative. The Internet, built on open standards, serves as the original Open Source venture. The success of the Internet and its open standards model is mirrored in the Open Source movement. The Internet has also provided a new infrastructure for Open Source to thrive, lowering barriers to entry and distribution costs.

1.9

The Open Source development model owes its success to the rapid dissemination of information facilitated by the Internet. Similar to the impact of the printing press on the Renaissance, the Internet has revolutionized the sharing of knowledge. Open Source is deeply rooted in the academic tradition of open sharing and community building. The Internet serves as the modern-day printing press, lowering barriers to entry and enabling instant distribution of source code. However, the computer industry must not forget its scientific heritage and continue to nourish computer science through the Open Source model. Industry benefits from the innovation and creativity that Open Source brings, as exemplified by the success of Linux. Open Source development drives progress in both computer science and the computer industry.

A Brief History of Hackerdom

2.1

The early days of computing saw the emergence of a group of programmers known as Real Programmers. They were the precursors to the hacker culture and were highly skilled and enthusiastic individuals who built and played with software for fun. Coming from engineering and physics backgrounds, they coded in machine language, assembler, and early programming languages like FORTRAN.

During the era of batch computing and mainframes, from the end of World War II to the early 1970s, Real Programmers dominated the technical culture in computing. They contributed to the formation of hacker folklore, including stories like the legendary Mel's story and the popular Murphy's Laws.

Some notable figures from the Real Programmer culture remained active well into the 1990s. Seymour Cray, the designer of Cray supercomputers, was renowned for his programming prowess and technical feats. Others, like Stan Kelly-Bootle, transitioned into writing and became folklorists, engaging with the hacker culture through technical humor columns.

However, the rise of interactive computing, universities, and networks brought significant changes to the programming landscape. These developments led to the evolution of a continuous engineering tradition, which eventually gave birth to today's open-source hacker culture. The culture of Real Programmers played a crucial role in laying the foundation for this transformation.

2.2

The hacker culture as we know it today can be traced back to 1961 when MIT acquired the first PDP-1 computer. The Signals and Power committee of MIT's Tech Model Railroad Club (TMRC) adopted the machine and created a surrounding culture that gave birth to programming tools, slang, and a distinct hacker identity. MIT's computer culture was the first to adopt the term "hacker."

The TMRC hackers from MIT became the nucleus of MIT's Artificial Intelligence Laboratory (AI Lab), which became a leading center for AI research. Their influence spread further after 1969 with the establishment of the ARPAnet, the first transcontinental, high-speed computer network. The ARPAnet connected universities, defense contractors, and research laboratories, enabling unprecedented exchange of information and collaboration.

The ARPAnet brought together hackers from across the United States, forming a networked tribe. It facilitated the propagation of hacker artifacts such as slang lists, satires, and discussions of the hacker ethic. The universities connected to the ARPAnet, particularly their computer science departments, played a significant role in the growth of hackerdom.

The hacker culture's fortunes were intertwined with Digital Equipment Corporation's (DEC) PDP series of minicomputers. DEC's machines, especially the PDP-10, which was released in 1967, were popular among hackers. The ARPAnet primarily consisted of DEC machines and played a crucial role in the development of hacker culture.

While most universities used DEC machines, MIT took a different path. They built their own operating system called ITS (Incompatible Timesharing System) for the PDP-10, rejecting DEC's software. ITS, written in assembler and utilizing the powerful LISP language, hosted numerous technical innovations and contributed to the successes of MIT's hackers.

Other important centers of hacker culture included Stanford University's Artificial Intelligence Laboratory (SAIL), Carnegie-Mellon University (CMU), and Xerox PARC. SAIL and CMU made significant contributions to the development of personal computers, software interfaces, expert systems, and industrial robotics. Xerox PARC produced groundbreaking hardware and software innovations, including the invention of the modern graphical user interface.

Throughout the 1970s, the ARPAnet and PDP-10 cultures thrived, and electronic mailing lists became popular for fostering cooperation and communication among hackers. DARPA, the Defense Department agency overseeing the ARPAnet, tolerated the "unauthorized" activities, understanding the value of attracting bright young people into the computing field.

The hacker culture's influence continued to grow, and its communication style laid the foundation for later commercialized services like CompuServe, GEnie, and Prodigy. The hacker culture, nurtured by the ARPAnet and driven by waves of technological change, set the stage for the open-source hacker culture that exists today.

2.3

In 1969, while the PDP-10 tradition was flourishing, a Bell Labs hacker named Ken Thompson invented Unix. Thompson had been involved in the development of Multics, a time-sharing operating system that aimed to simplify user interaction and programming. However, Bell Labs withdrew from the project when Multics became overly complex. Thompson, missing the Multics environment, started implementing his own ideas on a DEC PDP-7 computer.

Another hacker, Dennis Ritchie, created the programming language C specifically for use with Thompson's Unix. Both Unix and C were designed to be flexible and unconstraining. The tools gained popularity within Bell Labs, and in 1971, Thompson and Ritchie won a bid to develop an office-automation system. However, they had larger ambitions in mind.

Thompson and Ritchie realized that the hardware and compiler technology had advanced to the point where an entire operating system could be written in C. By 1974, Unix had been successfully ported to multiple machine types, marking a significant milestone. Unix presented the same face and capabilities across different machines, serving as a common software environment. This eliminated the need for users to invest in new software each time a machine became obsolete.

Unix and C had several strengths. They adhered to a "Keep It Simple, Stupid" philosophy, making the logical structure of C easily graspable. Unix was designed as a flexible toolkit of simple programs that could be combined in useful ways. This adaptability allowed Unix to be employed in various computing tasks beyond the designers' original intentions.

The popularity of Unix grew rapidly within AT&T and spread to numerous university and research computing sites by 1980. The PDP-11 and VAX were the workhorse machines of the early Unix culture, but due to Unix's portability, it ran on a wide range of machines. Unix sites began to form a network nation with their own hacker culture. Unix even had its own networking protocol called Unix-to-Unix Copy Protocol (UUCP), enabling point-to-point electronic mail exchange over ordinary phone lines.

While Unix and the PDP-10 cultures encountered each other on the periphery, they initially didn't mix well. PDP-10 hackers viewed Unix users as upstarts with primitive tools compared to the complexities of LISP and ITS. Additionally, a third current emerged with the rise of personal computers in the mid-1970s, attracting a new generation of hackers who used the BASIC language. Both PDP-10 enthusiasts and Unix aficionados considered BASIC beneath their consideration.

2.4

In 1980, three distinct hacker cultures emerged: ARPAnet/PDP-10, Unix/C, and microcomputers. The decline of PDP-10 and ITS was driven by aging technology and commercialization attempts, while Unix and the Berkeley variant gained dominance. Richard M. Stallman, a prominent figure, formed the Free Software Foundation and created a Unix clone, preserving ITS's spirit. The rise of microchips and local-area networks had a significant impact on hackerdom, with startups developing the first workstations. Sun Microsystems, founded in 1982 by Unix hackers from Berkeley, introduced affordable Unix-based systems that shaped the industry. Despite ITS's demise, its influence endured through Stallman's efforts and the evolving hacker culture around Unix. The stage was set for the rapid growth and transformation of the hacker community in the coming years.

2.5

By 1984, the hacker community was divided between the networked hackers centered around the Internet and Usenet, primarily using minicomputer or workstation-class machines running Unix, and a disconnected group of microcomputer enthusiasts. Workstations brought high-performance graphics and network capabilities, driving software and tool-building challenges. Berkeley Unix incorporated ARPAnet protocols, facilitating Internet growth. The X Window System prevailed over proprietary graphics systems, reflecting the hacker ethic of freely sharing sources. The ITS/Unix rivalry diminished as ITS faded away, and the focus shifted to the rivalry between Berkeley Unix and AT&T versions. Microcomputers based on Intel 386 chips threatened workstations, making powerful Unix-based machines affordable. However, the DOS and Mac hacker populations lacked a cohesive culture. Commercial Unixes remained expensive, prompting a search for affordable alternatives. The failure of commercializing proprietary Unix led to the rise of Microsoft's Windows OS. In the early 1990s, the future of Unix seemed uncertain, but hidden developments would soon bring about transformative successes.

2.6

In the midst of the HURD's failure, Linus Torvalds emerged in 1991 and started developing a free Unix kernel called Linux. Unlike traditional software development models, Linux attracted a large number of volunteers who coordinated through the Internet, releasing updates weekly and gathering feedback from users. This unconventional approach proved successful, and by 1993, Linux competed with commercial Unixes in stability, reliability, and software availability. The rise of Linux led to the decline of smaller commercial Unix vendors, while BSDI thrived by offering full sources with its BSD-based Unix and fostering connections with the hacker community. These developments went largely unnoticed outside the hacker culture, but they were gradually reshaping the commercial software world.

2.7

The early 1990s witnessed the growth of both Linux and the Internet. The Internet became more accessible to the public through the emerging Internet-provider industry, while the World Wide Web accelerated its expansion. By 1994, free Unix versions like Linux and 386BSD gained significant attention, with Linux being commercially distributed and widely adopted. Major computer companies started promoting their software and hardware as Internet-friendly. During this time, hackerdom focused on Linux development and the mainstreaming of the Internet. The Internet's growing popularity brought the hacker culture some mainstream respectability and political influence. Hacker activism successfully opposed government control of encryption through the Clipper proposal and fought against internet censorship through the Communications Decency Act. This period marks a shift from being an observer to becoming an actor in the narrative, leading to subsequent events in "The Revenge of the Hackers."

Twenty Years of Berkeley Unix

3.1

In November 1973, Ken Thompson and Dennis Ritchie presented the first Unix paper at the Symposium on Operating Systems Principles. Professor Bob Fabry of UC Berkeley expressed interest and obtained a PDP-11/45 to run Unix. In January 1974, Unix Version 4 was installed by graduate student Keith Standiford. Remote debugging became necessary, with Thompson calling Standiford to insert the phone into the modem due to limited connectivity. The cooperation between Berkeley and Bell Labs allowed for rapid software improvement. However, conflicts arose as the Math and Statistics departments wanted to run DEC's RSTS system. A compromise was reached, and Unix and RSTS were scheduled for eight-hour shifts, with students preferring to work on Unix despite the unusual schedule. Professors Eugene Wong and Michael Stonebraker moved their INGRES database project from the batch environment to the interactive Unix environment. They purchased an 11/40 running Unix Version 5 in 1974, becoming the first group in the Computer Science department to distribute their software. To address the shortage of machine time, Professors Stonebraker and

Bob Fabry acquired two instructional 11/45s in 1974. In 1975, they pooled the funds to purchase an 11/70, a superior machine. Ken Thompson joined UC Berkeley as a visiting professor, bringing Unix Version 6. Graduate students Bill Joy and Chuck Haley worked on improving the Pascal interpreter and developing the ex editor. With Thompson's departure in 1976, Joy and Haley began exploring the Unix kernel, suggesting enhancements to streamline kernel performance.

3.2

In 1977, Bill Joy created the first distribution of the Berkeley Software Distribution (BSD), which included the Pascal system and the editor ex. The distribution received requests, and about thirty free copies were sent out. With the availability of ADM-3a terminals, Joy developed the vi editor, providing screen-based editing. To consolidate screen management, Joy created a small interpreter driven by a terminal description, which later became termcap. In 1978, the software distribution was updated, resulting in the Second Berkeley Software Distribution (2BSD). It included an enhanced Pascal system, vi, and termcap for multiple terminals. Joy continued to handle distributions and incorporate user feedback. The 2BSD distribution expanded, and the final version, 2.11BSD, became a widely used system on numerous PDP-11 computers worldwide.

3.3

In 1978, Professor Richard Fateman sought a machine with a larger address space for his work on Macsyma and obtained a VAX-11/780. Although the VAX initially ran VMS, the department preferred Unix and acquired the 32/V port by John Reiser and Tom London. However, 32/V did not utilize the VAX's virtual memory capability. Fateman approached Professor Domenico Ferrari, who enlisted Ozalp Babaoglu to develop a virtual memory system for Unix on the VAX. Bill Joy joined the effort and helped integrate the code into 32/V. After debugging, 32/V was phased out, and Joy began porting 2BSD software to the VAX. By the end of 1979, the first distribution of 3BSD was shipped from Berkeley. With the commercialization of Unix, Bell Laboratories could no longer serve as a clearing-house for ongoing research, and Berkeley took on the role of producing research releases due to its early involvement and history with Unix-based tools.

3.4

In the late 1970s, DARPA (Defense Advanced Research Projects Agency) recognized the need to unify operating systems across its nationwide computer network. The high cost and challenges of porting research software to different machines prompted DARPA to choose Unix as a standard due to its proven portability. In response to DARPA's interest, Bob Fabry proposed that Berkeley develop an enhanced version of 3BSD for the DARPA community. The successful release of 3BSD in December 1979 paved the way for an 18-month contract with DARPA, leading to the formation of the Computer Systems Research Group (CSRG) at Berkeley. Bill Joy, expressing interest in leading the software development, became the project leader. The development of Unix at Berkeley continued, resulting in the release of 4BSD in October 1980, which included additional features and enhancements. A rebuttal to performance criticisms led to the tuning of the kernel, and the improved system was released as 4.1BSD in June 1981. The naming scheme for future releases was changed to stay at 4BSD to avoid confusion with AT&T's System V.

3.5

After the release of 4.1BSD, the success of the Unix project at Berkeley led to a new two-year contract with DARPA, providing increased funding for further development. The goals for the new system were set based on the needs of the DARPA research community, including a faster file system, support for large address spaces, flexible interprocess communication, and networking integration. A steering committee was formed to guide the design work and ensure the research community's needs were addressed.

In the summer of 1981, the implementation of the new file system began, and Bill Joy focused on developing prototype interprocess communication facilities. When TCP/IP protocols were integrated into the system, it became apparent that support for multiple network protocols was necessary, leading to internal restructuring. A preliminary system called 4.1a was distributed locally, followed by the creation of the "4.2BSD System Manual" that described the proposed user interfaces for the 4.2BSD system.

By June 1982, the new file system was fully integrated into the kernel, resulting in the 4.1b system. Bill Joy announced his departure to join Sun Microsystems, and Sam Leffler took over as the project lead. Deadlines were set, and an intermediate release called 4.1c was distributed in April 1983 to prepare for the final system release. In August 1983, the completed 4.2BSD system was released.

Following the completion of 4.2BSD, Mike Karels replaced Sam Leffler at the CSRG. The popularity of 4.2BSD was significant, with over 1,000 site licenses issued within eighteen months. Many Unix vendors opted to ship 4.2BSD rather than AT&T's System V due to its networking capabilities and the Berkeley Fast Filesystem. However, as networking and other improvements were integrated into System V, vendors gradually switched back to it while incorporating further BSD developments.

3.6

After two years of tuning and refining the system, Mike Karels and the team anticipated releasing 4.3BSD in the summer of 1985. However, this plan was halted when it was pointed out that the networking code in 4.2BSD had never been updated with the final version from BBN. BBN requested that Berkeley replace the TCP/IP code with their implementation. After evaluating both code bases, Mike Karels decided to incorporate the good ideas from the BBN code into the Berkeley code base, rather than replacing it entirely. DARPA initially suggested releasing both implementations but ultimately chose to stick with the Berkeley code base based on an evaluation by an independent third party.

The polished 4.3BSD system was released in June 1986, addressing many performance complaints. Although vendors had started switching back to System V, parts of 4.3BSD were incorporated into their systems, particularly the networking subsystem. In October 1986, Keith Bostic joined the CSRG and successfully ported 4.3BSD to the PDP-11, resulting in the 2.11BSD release.

As the VAX architecture reached its end, the CSRG began exploring other machines for running BSD. They considered the Power 6/32 architecture by Computer Consoles, Incorporated. While the architecture ultimately died, the CSRG received machines from the company to continue their work. They successfully split the BSD kernel into machine-dependent and machine-independent parts, leading to the release of 4.3BSD-Tahoe in June 1988. This work proved valuable as BSD was ported to numerous other architectures.

3.7

In June 1989, Berkeley released Networking Release 1, which included the TCP/IP networking code and utilities. This marked the first freely-redistributable code from Berkeley. Vendors could now obtain the networking code without requiring an expensive AT&T source license. The licensing terms were liberal, allowing modification and distribution without royalties. Although a fee was charged for the tape, copies were freely exchanged and available for anonymous FTP. The revenue from sales supported further development at the CSRG.

3.8

In the development of the base system, the CSRG integrated the virtual memory system from the MACH operating system and a Sun-compatible version of the Network Filesystem (NFS) developed by Rick Macklem. Despite not having the complete feature set of 4.4BSD ready, the CSRG decided to release an interim version called 4.3BSD-Reno in early 1990. This release aimed to gather feedback and experiences on the new additions. It was named after Reno, a gambling city in Nevada, as a playful reference to the experimental nature of the release.

3.9

Following the success of the Networking Release 1, Keith Bostic spearheaded an effort to rewrite the Unix utilities based on their published descriptions, resulting in the completion of over a hundred utilities and libraries. Mike Karels, Keith Bostic, and others then focused on removing proprietary code from the kernel, resulting in only six remaining contaminated files. The CSRG obtained permission to release the expanded code base under Networking Release 2, which shipped in June 1991. Bill Jolitz later released 386/BSD, a fully compiled and bootable system for the PC architecture, which gained a significant following. The NetBSD and FreeBSD groups were formed shortly after, with NetBSD emphasizing support for multiple platforms and research-style development, while FreeBSD targeted a larger user base with easier installation and CD-ROM distribution. OpenBSD spun off from NetBSD, focusing on system security and ease of use. These projects continue to thrive, offering different strengths and attracting diverse user communities.

3.10

Berkeley Software Design, Incorporated (BSDI) was established to develop and distribute a commercially supported version of the BSD code. However, they faced a legal challenge from Unix System Laboratories (USL), a subsidiary of AT&T, which demanded that BSDI stop promoting their product as Unix. USL filed a lawsuit against BSDI, alleging that the BSDI product contained proprietary code and trade secrets. The lawsuit eventually expanded to include the University of California as well. After a lengthy legal battle and a preliminary hearing, the judge dismissed most of the complaints and denied the injunction. USL refiled the suit against BSDI and the University of California, leading to further legal proceedings. In December 1992, the judge denied the injunction and narrowed the remaining complaints. The University of California filed a countersuit against USL in a California state court. Settlement talks began in 1993, and a settlement was reached in January 1994, resulting in the removal of three files from Networking Release 2 and minor changes to other files. The University agreed to add USL copyrights to certain files, while still allowing their free redistribution.

3.11

The settlement of the lawsuit led to the release of 4.4BSD-Lite in June 1994. It had the same redistribution terms as the Networking releases, allowing free distribution of source and binary code with the requirement to maintain University copyrights and provide credit. Simultaneously, a version called 4.4BSD-Encumbered was released, requiring a USL source license.

The settlement also mandated that organizations using 4.4BSD-Lite as their base would not be sued by USL. Consequently, BSD groups like BSDI, NetBSD, and FreeBSD had to restart their code base with 4.4BSD-Lite, resulting in a temporary delay but allowing them to incorporate the three years of development from the CSRG.

3.12

After the release of 4.4BSD-Lite Release 2 in June 1995, the CSRG disbanded, passing the torch to new groups with fresh ideas. The decentralized nature of the BSD development allowed for diverse approaches and easy adoption of the best ideas across different groups. The open source software movement, which includes the well-known Linux system, has gained significant attention and respect. Approximately half of Linux's utilities come from the BSD distribution, and the development tools from the Free Software Foundation are crucial for Linux distributions. Together, the CSRG, Free Software Foundation, and Linux kernel developers have played a pivotal role in launching the Open Source software movement. The goal is to see Open Source become the preferred method for software development and acquisition by users and companies worldwide.

The Internet Engineering Task Force

4.1

The IETF (Internet Engineering Task Force) was established in January 1986 as a quarterly meeting of U.S. government-funded researchers. Non-government vendors were later invited to participate, starting from the fourth meeting in October of that year. Over time, the IETF meetings became open to anyone interested in attending. The early meetings had a small number of attendees, with the peak attendance of 120 people at the 12th meeting in January 1989. However, the IETF has experienced significant growth in attendance since then, with more than 2,100 attendees at the 43rd meeting in December 1998.

In 1991, the number of IETF meetings per year was reduced from four to three. The IETF operates with a small Secretariat based in Reston, VA, and an RFC Editor managed by the University of Southern California's Information Sciences Institute.

The IETF itself has not been legally incorporated but rather functions as an activity without legal substance. Initially, its expenses were covered by U.S. government grants and meeting fees. However, since the beginning of 1998, the expenses have been funded by meeting fees and the Internet Society.

The Internet Society, formed in 1992, serves as a legal umbrella over the IETF standards process and provides funding for IETF-related activities. It is an international membership-based non-profit organization that also promotes the Internet in regions where it is not yet widely accessible. The IETF operates as a standards development function under the auspices of the Internet Society.

At the 5th IETF meeting in February 1987, the concept of working groups was introduced, and currently, there are over 110 working groups within the IETF.

4.2

The IETF is a membership organization without defined criteria for membership. Any individual participating in IETF mailing lists or attending IETF meetings is considered a member. There are currently 115 working groups organized into eight areas, each managed by volunteer Area Directors. The Area Directors, along with the IETF chair, form the Internet Engineering Steering Group (IESG), which serves as the standards approval board. The IAB provides advice on working group formation and architectural implications. Members of the IAB and Area Directors are selected for two-year terms by a nominations committee randomly chosen from volunteers who have attended previous IETF meetings.

4.3

The IETF operates in a bottom-up manner, where most working groups are formed by individuals who propose the group to an Area Director. The IESG and IAB rarely create working groups on their own. When a working group is proposed, the Area Director works with the proposers to develop a charter that outlines the group's specific deliverables, potential liaisons, and limitations. The charter is then circulated for comments from the IESG, IAB, and other stakeholders. After the necessary approvals, the working group is officially created. This approach ensures enthusiasm and expertise within the working groups while allowing for public input and consideration of related work in other organizations.

4.4

The IETF publishes documents called RFCs, which are freely available on the Internet. RFCs fall into two categories: standards track and non-standards track. The IETF holds a limited copyright on RFCs to ensure their availability and the creation of derivative works. Authors retain their rights. The RFC series is the main publication platform. Internet-Drafts, temporary documents, serve as works in progress and are not for citation or reference.

4.5

The IETF operates under the motto "rough consensus and running code." Working group unanimity is not required, but proposals must demonstrate significant support. The IETF does not have fixed percentage requirements for approval, but proposals with over 90% support are more likely to be approved. Non-standards track documents can come from working groups or individuals, and most proposals are reviewed by the IESG. The IESG evaluates proposals and can approve or request revisions. The standards track progresses from Proposed Standard to Draft Standard to Internet Standard, with requirements such as clear documentation and multiple independent implementations. The IETF process emphasizes bottom-up task creation and values practical implementation.

4.6

The IETF's open documentation and standards development policies have contributed to its success. Unlike many other standards organizations, the IETF makes all documents, mailing lists, and meetings openly available. This openness allows for broader participation and prevents restrictions on access. In contrast, restricted participation and limited document access can lead to standards that don't meet user needs or are overly complex. The IETF has supported the concept of open sources even before the formal Open Source movement emerged. The availability of open standards processes and documentation is essential for the success of projects like the Open Source movement. This partnership between open standards, documentation, and sources has driven the development of the Internet and will continue to foster innovation in the future.

The GNU Operating System and the Free Software Movement

5.1

In 1971, when I joined the MIT Artificial Intelligence Lab, I became part of a software-sharing community that had been around for years. We shared software freely, which was not uncommon in the computer world. At the AI Lab, we used a time-sharing operating system called ITS that we had designed and written ourselves. As system hackers, our goal was to improve this system. Although we didn't call it "free software" at the time, that's essentially what it was. If someone from another university or company wanted to use our programs, we gladly allowed them to port and utilize them. If you came across an interesting program being used by someone, you could simply ask to see the source code and learn from it, modify it, or even use parts of it for your own programs.

5.2

In the early 1980s, the situation changed dramatically when Digital discontinued the PDP-10 series, rendering the programs built for the ITS operating system obsolete. The AI Lab hacker community had already collapsed due to the hiring of its members by Symbolics, a spin-off company. The AI Lab, now depopulated, decided to use Digital's non-free timesharing system instead of ITS.

During this time, modern computers like the VAX or the 68020 had their own operating systems, but none of them were free software. Users had to sign nondisclosure agreements just to obtain executable copies, which meant that sharing or collaborating with others was prohibited. Proprietary software companies enforced a social system where users were helpless and could only request changes from the software owners.

The notion that this proprietary software social system, which restricts sharing and modifying software, is unethical and wrong may be surprising to some readers. However, it is based on the premise that dividing the public and keeping users powerless is an unjust practice. Software publishers have worked hard to establish this perspective as the only valid one.

When software publishers talk about "enforcing rights" or "stopping piracy," the unstated assumptions underlying these statements are crucial. One assumption is that software companies have an inherent and unquestionable right to own software and exert control over its users. Another assumption is that the sole importance of software lies in the tasks it enables, disregarding the societal implications. A third assumption is that usable software cannot exist without granting companies power over users, although the free software movement has demonstrated otherwise.

By rejecting these assumptions and prioritizing the users' interests and common-sense morality, we reach different conclusions. Users should be free to modify programs to suit their needs and share software because helping others is fundamental to society. For a more comprehensive exploration of the reasoning behind these conclusions, refer to the web page http://www.gnu.org/philosophy/why-free.html.

5.3

Facing the moral dilemma of joining the proprietary software world or leaving the computer field altogether, the author, Richard Stallman, made a different choice. He recognized the need to create an operating system that would enable a community of cooperating hackers and allow anyone to use a computer without restricting their freedom.

With his skills as an operating system developer, Stallman took on the task of creating a free operating system compatible with Unix. The project was named GNU, a recursive acronym for "GNU's Not Unix." Stallman aimed to develop a complete operating system that included essential components such as command processors, compilers, interpreters, and text editors.

The decision to start the GNU project was driven by a sense of responsibility and the belief that if he didn't take action, no one else would. Stallman's choice was guided by the principle of not only advocating for oneself but also considering the larger community. The quote attributed to Hillel encapsulates this sentiment: "If I am not for myself, who will be for me? If I am only for myself, what am I? If not now, when?"

5.4

The term "free software" refers to freedom rather than price. It is important to understand the definition of free software in this context:

Freedom to run the program for any purpose. Freedom to modify the program and access its source code. Freedom to redistribute copies, either for free or for a fee. Freedom to distribute modified versions of the program. The freedom to sell copies of free software is crucial because it supports the development of free software and allows for the creation of collections sold on CD-ROMs. Any program that restricts these freedoms is not considered free software.

While there have been attempts to find alternative terms for "free," none have been able to capture the exact meaning of freedom in the context of software. Words like "liberated," "freedom," and "open" either have different meanings or limitations.

In summary, free software is about the essential freedoms that users have to run, modify, distribute, and improve software, regardless of its price.

5.5

In the process of developing the GNU system, I made use of existing pieces of free software to make the project more feasible. For instance, I incorporated TeX as the main text formatter and later opted to utilize the X Window System instead of creating a new one for GNU.

It's important to note that the GNU system comprises not only GNU software but also programs developed by other individuals and projects for their own needs. These external programs are included in the GNU system because they are also free software and align with the principles of the project.

5.6

In January 1984, I made the decision to leave my job at MIT in order to focus on developing GNU software. This step was necessary to ensure that MIT would not have control over the distribution and terms of GNU as proprietary software. By leaving MIT, I could maintain the freedom and intention of creating a software-sharing community.

Fortunately, Professor Winston, who was the head of the MIT AI Lab at the time, kindly allowed me to continue using the Lab's facilities, which provided me with the necessary resources to carry out the development of GNU.

5.7

Shortly before starting the GNU project, I came across the Free University Compiler Kit (VUCK), a compiler designed for multiple languages and target machines. However, the author denied GNU the permission to use it, stating that the compiler was not free despite the university's name.

In response, I decided to develop my own multi-language, multi-platform compiler as the first program for the GNU project. I initially tried modifying the Pastel compiler, which supported an extended version of Pascal, but encountered limitations due to stack space constraints on the available 68000 Unix system.

Upon further analysis, I realized the Pastel compiler had limitations in its approach, parsing the entire input file and generating the output file without freeing storage. Consequently, I made the decision to write a new compiler from scratch. This led to the creation of GCC (GNU Compiler Collection), which eventually became a widely-used compiler suite. However, before diving into GCC, I focused on developing GNU Emacs.

5.8

In September 1984, I began working on GNU Emacs, and by early 1985, it was becoming usable. This allowed me to start using Unix systems for editing, as I had previously relied on other machines for my editing needs due to my lack of interest in learning vi or ed.

As interest grew in GNU Emacs, the question arose of how to distribute it. I initially made it available on the anonymous ftp server on the MIT computer I used, which became the main GNU ftp distribution site. However, not everyone had access to the internet at that time, so I needed to find a solution for those individuals.

I could have suggested that they find a friend on the net to make a copy for them, or followed the approach I used for the original PDP-10 Emacs, where users would mail me a tape along with a self-addressed stamped envelope (SASE), and I would mail it back with Emacs on it. However, since I was unemployed and looking for ways to generate income from free software, I announced that I would send a tape to anyone interested, charging a fee of $150. This marked the beginning of a free software distribution business, which served as a precursor to the companies that now distribute complete Linux-based GNU systems.

5.9

If a program is free software when it leaves the hands of its author, this does not necessarily mean it will be free software for everyone who has a copy of it. For example, public domain software (software that is not copyrighted) is free software; but anyone can make a proprietary modified version of it. Likewise, many free programs are copyrighted but distributed under simple permissive licenses that allow proprietary modified versions. The paradigmatic example of this problem is the X Window System. Developed at MIT, and released as free software with a permissive license, it was soon adopted by various computer companies. They added X to their proprietary Unix systems, in binary form only, and covered by the same nondisclosure agreement. These copies of X were no more free software than Unix was. The developers of the X Window System did not consider this a problem — they expected and intended this to happen. Their goal was not freedom, just "success," defined as "having many users." They did not care whether these users had freedom, only that they should be numerous. This lead to a paradoxical situation where two different ways of counting the amount of freedom gave different answers to the question, "Is this program free?" If you judged based on the freedom provided by the distribution terms of the MIT release, you would say that X was free software. But if you measured the freedom of the average user of X, you would have to say it was proprietary software. Most X users were running the proprietary versions that came with Unix systems, not the free version.

5.10

The goal of GNU was to give users freedom. Copyleft uses copyright law, but flips it over to serve the opposite of its usual purpose: instead of a means of privatizing software, it becomes a means of keeping software free.For an effective copyleft, modified versions must also be free.Therefore, a crucial requirement for copyleft: anything added to or combined with a copylefted program must be such that the larger combined version is also free and copylefted.

5.11

The Free Software Foundation accepts donations, but most of its income has always come from sales — of copies of free software, and of other related services. Today it sells CD-ROMs of source code, CDROMs with binaries, nicely printed manuals.Our goal was a complete operating system, and these programs (C compiler and bash) were needed for that goal.

5.12

The free software philosophy rejects a specific widespread business practice, but it is not against business. When businesses respect the users' freedom, we wish them success.

5.13

The principal goal of GNU was to be free software. Even if GNU had no technical advantage over Unix, it would have a social advantage, allowing users to cooperate, and an ethical advantage, respecting the user's freedom.

5.14

As the GNU project's reputation grew, people began offering to donate machines running Unix to the project. These were very useful, because the easiest way to develop components of GNU was to do it on a Unix system, and replace the components of that system one by one.

5.15

As the GNU project progressed, the GNU task list was created to identify missing components and recruit developers to fill those gaps. While many Unix components have been completed, the list now includes various applications and games to make a complete operating system. Compatibility with Unix games was not a concern, so a diverse range of games was listed to cater to user preferences.

5.16

The GNU C library is licensed under the GNU Library General Public License (LGPL), allowing it to be linked with proprietary software. This strategic decision aims to encourage the use of the GNU system by not restricting the compilation of proprietary programs. While there is no ethical obligation to allow proprietary applications, it is considered strategically advantageous to foster the development of free applications. For other libraries, the decision to use the GPL or LGPL depends on the specific case and whether it provides an advantage for free software developers. The goal is to create a collection of GPL-covered libraries that offer unique benefits for further free software development.

5.17

Eric Raymond says that "Every good work of software starts by scratching a developer's personal itch." Maybe that happens sometimes, but many essential pieces of GNU software were developed in order to have a complete free operating system. They come from a vision and a plan, not from impulse.

5.18

At the beginning of the GNU project, I imagined that we would develop the whole GNU system, then release it as a whole. That is not how it happened. Since each component of the GNU system was implemented on a Unix system, each component could run on Unix systems, long before a complete GNU system existed

5.19

By 1990, the GNU system was almost complete except for the kernel. The decision was made to develop the kernel using the Mach microkernel as a base, and the GNU HURD was created as a collection of servers running on top of Mach. The choice of this design aimed to avoid the challenge of debugging a kernel without a source-level debugger. However, the debugging process proved to be difficult and time-consuming, particularly due to the complexity of multithreaded servers communicating with each other. As a result, the development and stabilization of the HURD have taken many years.

5.20

The original name for the GNU kernel was Alix, named after my sweetheart at the time. However, it was later changed to HURD by the main developer of the kernel, Michael Bushnell (now Thomas). The name Alix was redefined within the kernel to refer to a specific part of the system. Eventually, the design changed, and Alix component disappeared. Although Alix and I broke up, her friend came across the name Alix in the HURD source code, fulfilling its purpose.

5.21

The GNU HURD is not ready for production use. Fortunately, another kernel is available. In 1991, Linus Torvalds developed a Unix-compatible kernel and called it Linux. Around 1992, combining Linux with the not-quite-complete GNU system resulted in a complete free operating system.

5.22

We have proved our ability to develop a broad spectrum of free software. This does not mean we are invincible and unstoppable. Several challenges make the future of free software uncertain; meeting them will require steadfast effort and endurance, sometimes lasting for years.

5.23

Hardware manufacturers' increasing trend of keeping hardware specifications secret poses challenges for developing free drivers to support new hardware in Linux and XFree86. To address this, programmers can resort to reverse engineering, although it is a demanding task. Another approach is for users to choose hardware that is supported by free software, creating a demand that discourages secrecy. The success of these strategies relies on a strong belief in the principles of free software and a widespread commitment to prioritize freedom over convenience or cost.

5.24

The presence of non-free libraries on free operating systems poses a trap for free software developers. These libraries attract developers with their features but create a dependency that hinders integration into a free operating system. The Motif toolkit and later Qt are examples of such traps. In response, the free software community developed alternatives like LessTif, GNOME, and Harmony to provide free replacements. The community's commitment to freedom and avoidance of non-free software has been critical. The challenge remains for the community to stay vigilant and prioritize freedom over convenience to avoid future traps and ensure the future of free software.

5.25

The worst threat we face comes from software patents, which can put algorithms and features off-limits to free software for up to twenty years. Those of us who value free software for freedom’s sake will stay with free software anyway. But those who value free software because they expect it to be technically superior are likely to call it a failure when a patent holds it back

5.26

The lack of good free manuals is a significant deficiency in our free operating systems. Free documentation, like free software, is about freedom, not price. A free manual must allow redistribution, both online and in print, and permission for modification. The freedom to modify documentation is crucial for free software because conscientious programmers can provide accurate and usable documentation alongside modified programs. Certain limits on modifications, such as preserving the original copyright notice or non-technical sections, are acceptable. However, all technical content must be modifiable and distributable without restrictions. The awareness and determination of free software developers will determine our ability to produce a comprehensive range of free manuals.

5.27

Estimates today are that there are ten million users of GNU/Linux systems such as Debian GNU/Linux and Red Hat Linux. But interest in the software is growing faster than awareness of the philosophy it is based on, and this leads to trouble. The efforts to attract new users into our community are far outstripping the efforts to teach them the civics of our community. We need to do both, and we need to keep the two efforts in balance.

5.28

Teaching new users about freedom became more difficult in 1998, when a part of the community decided to stop using the term “free software” and say open-source software” instead. “Free software” and “Open Source” describe the same category of software, more or less, but say different things about the software, and about values. The GNU Project continues to use the term “free software,” to express the idea that freedom, not just technology, is important

5.29

Yoda's philosophy may not work for everyone, including myself. I've often felt anxious and uncertain about my abilities, but I try anyway because there's no one else to protect what I care about. Sometimes I succeed, surprising myself, and other times I fail. But I continue to fight, finding new battles to defend what I believe in. It's a relief to see other hackers joining the cause, but the dangers we face are growing, with Microsoft now targeting our community. We can't take freedom for granted; we must be prepared to defend it.

Future of Cygnus Solutions

6.1

In 1989, after receiving approval from the California Department of Corporations, Cygnus Support was founded. The company's inception was driven by a vision that originated from reading Richard Stallman's GNU Emacs Manual. The manual, along with the GNU Manifesto, inspired a commitment to the principles of freely redistributable software and sharing among users. Cygnus Support aimed to provide support and services for GNU software, marking the beginning of a journey that continues today.

Stallman's GNU Manifesto presented a vision that went beyond its surface appearance of a socialist polemic. It outlined a business plan in disguise, emphasizing the unifying power of open source software and the potential for commercial services around it. The release of revolutionary programs like GNU Debugger (GDB) and GNU C Compiler (GCC) further showcased the superiority of open source tools over proprietary alternatives. The scalability, code quality, and rapid improvement of these tools hinted at the economic benefits of replacing proprietary technology with open source solutions.

The GNU Manifesto, while primarily focused on ethical imperatives, sparked a debate between the author and the speaker. The speaker reached a different conclusion, asserting that the freedom to use, distribute, and modify software will prevail for competitive, market-driven reasons rather than purely ethical ones. Initially, attempts were made to convince others of the merits of free software based on its benefits such as innovation and cost-effectiveness. However, a realization occurred that if everyone thought it was a great idea and no one believed it would work, there would be no competition. This led to the decision to start a company based on the commercial support of open-source software, as it seemed like an idea whose time had come.

In 1989, the state of proprietary software tools for programmers was dismal. They were primitive, had limitations, lacked proper support, and tied users to specific platforms. This demonstrated that free market economics were not effectively at work in the software marketplace. Understanding the flaws of the proprietary software model became crucial as it highlighted the need for a more true free market system. The limitations and shortcomings of proprietary software made the study of that model highly valuable in recognizing the potential for economic prosperity through freedom in software. The traditional model of proprietary software companies restricts the use and distribution of software through license agreements, patents, and trade secrets. By focusing on the micro level of software development and neglecting the macro level of freedom, the software industry suffered. Support for software was seen as an afterthought, and the absence of source code limited users' ability to implement desired features. Free market economics were turned upside down, leading to frustration and limited innovation. To challenge this model, starting a company that supported users at the source code level was necessary. While the initial cost of developing software was high, the long-term value and impact of open-source programs were significant. The analogy to common law and legal practice showcased the value of freely available precedents and standards. Just as lawyers command high value despite the availability of common law, creating and maintaining open-source standards held great potential for success. The goal was to create high-quality open-source programs that would become the de facto standards in the software world.

6.2

Despite lacking business experience, the founders of Cygnus embarked on putting their theory into practice. They used books from Nolo Press to incorporate the business and establish necessary formalities. However, they faced challenges and made costly mistakes along the way, learning from their lack of expertise in legal and corporate matters. Creating their own business model, they developed concepts in finance, accounting, marketing, sales, customer information, and support. As chaos ensued in the early stages, their focus was on providing technical support for proven software and leveraging economies of scale to make it profitable. They aimed to deliver higher quality support and development capabilities at a fraction of the cost compared to inhouse resources. In February 1990, they wrote their first contract, quickly followed by numerous others, signaling the reality of their business. By the end of the first year, they had secured significant support and development contracts. However, their success raised concerns about the scalability of the business and the need for operational and financial models. Overall, they faced challenges, made mistakes, but also achieved early success. The journey of turning theory into practice required continuous learning and adaptation to the evolving needs of their growing business.

6.3

In order to achieve economies of scale and prevent burnout, the founders of Cygnus decided to narrow their focus on a specific set of open-source technology that they could sell as a useful solution. They chose to sell the GNU compiler and debugger as a shrink-wrapped product, aiming to dominate the 32-bit compiler market. The GNU compiler already had broad support for various host environments and target architectures, making it one of the most widely ported compilers at the time. However, transitioning from GCC Version 1 to Version 2 proved challenging, as new optimizations were needed to compete on RISC platforms. The complexity of evaluating tradeoffs and the evolving nature of C++ posed additional difficulties, with the founders struggling to keep GNU C++ current amidst the distractions of running the business. The reality of creating a shrink-wrapped product turned out to be more demanding than anticipated, requiring extensive testing, product collateral, and careful consideration of the evolving market dynamics. Nonetheless, they remained determined to succeed and achieve the scale necessary to pursue their broader vision of an open-source play in the software industry. Cygnus faced several challenges, including the fragmentation of the GNU Debugger (GDB), the lack of a complete toolchain, the absence of a C library for embedded systems, competition from established players, and skepticism about the need for their support business. However, the founders remained determined to overcome these obstacles and pursue their vision of providing high-quality open-source software and support services. With determination and a sense of urgency, the founders of Cygnus took on the challenge of completing the work for GCC 2.0 and G++ while simultaneously growing the company's top line. Each founder took on specific responsibilities, and new hires were brought in to support the development and testing efforts. Despite the initial estimate of six months, the project grew in scope, and some team members became bored with the strict product focus. However, the founders remained committed to their vision and continued to push forward. John Gilmore's call for contributions to GDB resulted in the collection of 137 versions of the software, which he used to design the architecture for GDB 4.0. The team embraced the challenge and remained determined to prove that it could be done. The founders faced the complex task of creating a single library to handle multiple binary file formats. Despite initial skepticism, Gumby designed the BFD library to achieve consistency and scalability. As engineering focused on development, the sales team faced pressure to secure contracts. Tensions arose between sales and engineering as open-source community support seemed lacking. After a year and a half, the first "Progressive Release" was achieved, supporting Sun3 and Sun4 platforms. Although the journey took longer than expected, the tools improved and infrastructure laid the groundwork for supporting diverse platforms, providing new opportunities for the company's future. In 1991, a business student joined the sales team and quickly became a strong advocate of the Open Source approach. Her ability to explain the complexity of the work and the business value of the software helped customers understand why they should choose Cygnus. Instead of selling the superiority of Cygnus engineers, she emphasized how their engineers would benefit from Cygnus' expertise in baseline porting, support, and maintenance. This combination of technical prowess and business benefits led to significant sales achievements. Moreover, this effort resulted in the development of new technologies like GNU configure, autoconf, automake, DejaGNU, GNATS, and more, which became standards in their own right. The GNUPro toolkit has achieved remarkable success, supporting over 175 host/target combinations and dominating the market. Competitors have attempted to offer commercial support for GNU software, but Cygnus maintains its position as the primary source of "true GNU" support due to its large team of skilled engineers who are heavily involved in the software's development. While competitors may add incremental features, the open-source nature of the software ensures that any value they contribute ultimately benefits Cygnus. This dynamic, unlike the win/lose competition in proprietary software, allows Cygnus to continuously benefit from its first-mover advantage established in 1989, giving it a ten-year head start over the competition.

6.4

Selling the merits of open-source software posed challenges as skeptics questioned the sanity, scalability, sustainability, profitability, manageability, and investibility of the model. Mainstream customers hesitated to adopt an unknown solution, despite testimonials, technical superiority, and lower prices. Furthermore, doubts arose regarding Cygnus' ability to scale its support business. While hiring engineers was successful, attracting capable managers proved difficult due to their shared concerns and prejudices. The company recognized the need for managers who could adapt to both open and closed source products. Despite these challenges, the Open Source model demonstrated resilience, with a high annual renewal rate and a customer-centric approach ensuring customer satisfaction.

6.5

In the embedded systems market, a few companies dominate the semiconductor and OEM sectors, while numerous small software companies offer their solutions. Fragmentation, redundancy, and high costs plague the market, hindering the acceleration of time to market. Recognizing the need for a silicon abstraction layer, Cygnus identified an opportunity to expand its product offering. They aimed to create a configurable, royalty-free real-time operating system (RTOS) to consolidate the market. However, the challenge of monetization remained uncertain. Despite the uncertainties, Cygnus made assumptions and outlined the necessary steps to address customer needs, including developing new configuration technology and completing the system on time. The cost of software development was a significant consideration in this endeavor. Despite initial assumptions that venture capitalists (VCs) wouldn't understand their business model, Cygnus was proven wrong. In 1992, Philippe Courtot, the first outside board member, introduced them to leading VCs who were impressed by their self-funding success and business model. However, as Cygnus expanded beyond GNUPro, they recognized the need for a new plan and partners. Greylock Management and August Capital invested $6.25M, the largest private placement for a software company in the first half of 1997. The technical and business teams faced challenges aligning the architecture of eCos with a commercialization strategy until they realized they were creating the world's first Open Source architecture. By empowering mainstream developers with high-level tools and demonstrating a 10x performance advantage over competitors, Cygnus positioned eCos as a game-changer for the embedded systems market.

6.6

Open-source software operates as an efficient free market, guided by Open Source businesses like an "invisible hand" following Adam Smith's concept. These businesses aim to benefit both the market and achieve their own goals. The Internet, fueled by open-source software, has played a pivotal role in enabling its development. As people continue to connect and engage with Open Source, we can expect revolutionary changes in software development, paralleling the transformative impact of the Renaissance on academic knowledge. Open-source software grants freedoms that have the potential to reshape the future of technology, fostering innovation and advancements beyond imagination.

Software Engineering

7.1

The software engineering process includes elements such as Marketing Requirements, System-Level Design, Detailed Design, Implementation, Integration, Field Testing, and Support. Each element should be completed before moving on to the next, and changes should trigger reviews of dependent elements. Advanced development may involve implementing a module before its dependencies are fully specified. Reviews, including peer, mentor/management, and cross-disciplinary reviews, are important throughout the process. Elements should have version numbers and auditable histories, and changes should undergo appropriate review based on their scope.

- Marketing Requirements:

  The software engineering process begins with creating a Marketing Requirements Document (MRD). This document outlines the target customers, their needs, and the product features that address those needs. It is important for engineering to be involved in this process to ensure feasibility and avoid unrealistic expectations. The MRD should be a collaborative effort between marketing and engineering.

- System-Level Design:

  The system-level design document provides a high-level description of the product, including its modules and their interaction. Its purpose is to assess the feasibility of the product and estimate the required effort for development. Additionally, the document outlines the system-level testing plan, ensuring that customer needs are addressed by the proposed design.

- Detailed Design:

  Detailed design documentation often becomes neglected in unfunded and informal projects where the focus is on having fun. While some individuals may enjoy working on detailed designs, it is often treated as an afterthought during implementation. External API documentation, such as header files or manpages, may also be overlooked if not intended for wider use. This is unfortunate, as valuable and potentially reusable code can remain hidden. Even modules specific to a project should ideally have documentation, benefiting others who may want to enhance or understand the codebase.

- Implementation

  Implementation is the enjoyable part of software development that programmers truly love. It allows them to express themselves freely and experiment with different coding styles, memory optimizations, and performance enhancements. Open-source projects provide a platform for programmers to try out new approaches and create beautiful code artifacts. In unfunded Open Source efforts, there is often flexibility in rigor and consistency, as users primarily care about functionality. However, developers themselves may value code review and gradually learn to appreciate it. Informal review processes and the absence of unit tests are common in such projects.

- Integration

  During the integration phase of an open-source project, several tasks are typically undertaken. These include creating manpages, ensuring the project builds successfully on various systems, cleaning up the

Makefile, writing a README file, packaging the project into a tarball, and making it available for download via anonymous FTP. Announcements about the project can be shared on mailing lists or newsgroups like comp.sources.unix. System-level testing is usually minimal or absent, as there is often no formal test plan or unit tests. While open-source efforts may lack extensive pre-release testing, this is not considered a weakness, as explained further below.

- Field Testing

  Unfunded open-source software benefits from extensive system-level testing, often surpassing the testing efforts of commercial software. This is due to the collaborative nature of the open-source community, where users, including developers themselves, actively engage in testing and debugging the source code. Users' friendly and helpful nature, combined with their ability to read and modify the source code, leads to valuable real-world experiences and feedback. The peer review process further enhances the testing effectiveness, as numerous programmers examine the code for bugs and potential security flaws. Although the risk of undisclosed security flaws exists, the overall advantage of having a large community of contributors reviewing the source code outweighs this concern and keeps open-source developers vigilant and accountable.

- Support

  The open-source development process relies on user feedback and community contributions to identify and fix bugs. Users play a crucial role by reporting issues and, in some cases, even providing patches to address them. This decentralized approach to bug triage can appear chaotic but is effective in identifying and resolving software flaws. While the lack of official support may deter some users, it also creates opportunities for consultants and software distributors to offer support contracts and commercial versions. Traditional software houses, including Unix vendors, have struggled to adapt to this model and often require additional quality assurance processes or revised support agreements to accommodate the open-source nature of the software. Ultimately, the success of the software support market will depend on harnessing the expertise and contributions of the open-source community, as it consistently delivers functionality that users desire, as seen in the comparison between Linux and Windows.

7.4

Engineering, whether in software, hardware, or other fields, follows a similar process. It involves identifying requirements, designing solutions, modularizing the design, implementing it, and performing testing, delivery, and support. Different fields may emphasize certain phases more than others. The transition from a programmer to a software engineer occurs when one recognizes the need for a different mindset and a greater focus on engineering principles. Open-source developers often experience this realization later because the collaborative nature of Open Source projects can delay the impact of the lack of engineering rigor. Understanding software engineering is essential for Open Source programmers to create high-quality systems. Software engineering encompasses proven techniques that go beyond the approach of individual programmers. It is a rich field that combines the best practices of the past and present to build robust systems.

The Linux Edge

8.1

Linux is not a version of Unix but aims to provide compatibility with Unix. Unlike FreeBSD, which is directly descended from the source code of Berkeley Unix, Linux was written from scratch without reference to Unix source code. Therefore, Linux is considered a new operating system.

Initially, the focus of Linux development was to create an operating system that would run on the 386 architecture. The first successful port of the Linux kernel was to the Motorola 68K series, which was used in early Sun, Apple, and Amiga computers. However, this port was not considered a true success as it followed a similar approach to the original Linux development, writing code from scratch targeted for a specific interface.

The need for portability became evident when efforts were made to port Linux to the DEC Alpha machine. The experience with the 68K port led to the realization that maintaining multiple code bases for different architectures would be unmanageable. As a result, the Linux kernel underwent a major rewrite to create a common code base that could support different architectures while managing development effectively.

This shift in development allowed Linux to work with a growing community of developers and ensure a more portable and manageable code base for supporting multiple architectures.

8.2

When I started working on the Linux kernel in 1991, there was a prevailing belief that portability could be achieved through a microkernel approach. Microkernels focused on abstracting details of process control, memory allocation, and resource allocation, pushing system specifics into user space. However, I found microkernels to be experimental, more complex than monolithic kernels, and slower in execution.

Speed was crucial to me, and I believed that many optimization tricks applied to microkernels could also be applied to traditional monolithic kernels. This led me to view the microkernel approach as essentially dishonest, driven by the pressure in the research community to pursue it for funding. Even the team designing Windows NT felt compelled to pay lip service to the idea of a microkernel.

Fortunately, I didn't face much pressure to pursue microkernels. The University of Helsinki, where I was working, had a different perspective on operating system research. They saw the operating system kernel as a well-understood topic, and the basics of operating systems were established by the early 1970s.

In my view, code can be made portable by programming intelligently rather than relying on abstraction layers. Trying to make microkernels portable is a waste of time because it abstracts away the crucial aspect that needs to be highly efficient - the kernel itself.

The goals of microkernel research aimed for a theoretical ideal, seeking maximum portability across any architecture. In contrast, my focus with Linux was on real-world systems' portability rather than theoretical systems.

8.3

The Alpha port of Linux began in 1993 and took about a year to complete. Although it wasn't fully finished at that time, it laid the foundation for design principles that have guided Linux and made subsequent ports easier.

The Linux kernel is not written with the goal of being portable to every architecture. Instead, I focused on supporting architectures that followed certain basic rules and were fundamentally sound. For example, while memory management can vary across different machines, there are commonalities in the use of paging and caching. By designing the Linux kernel's memory management to a common denominator among these architectures, it became easier to modify the code for the specific details of a particular architecture.

A few assumptions greatly simplified the porting process. For example, if a CPU supports paging, it must have some form of translation lookup buffer (TLB) to map virtual memory. While the exact form of the TLB may vary, the important aspects are how to fill it and how to flush it. So, in a sane architecture, you need a few machine-specific parts in the kernel, while most of the code is based on general mechanisms.

I also follow the principle of using compile-time constants instead of variables whenever possible. This allows the compiler to optimize the code more effectively. By setting up code to be flexibly defined yet easily optimized, better performance can be achieved.

An interesting aspect of this approach is that by defining a sane common architecture, you can present a better architecture to the operating system than what is actually available on the hardware platform. This may seem counterintuitive, but it allows for generalizations that align with performance optimizations. When making decisions based on observations and surveys of different systems, the same conclusions are often reached that would have been made solely for optimizing the kernel on a specific architecture. This convergence demonstrates that principles of portability and optimization often align.

By mixing the best of theory with the practical considerations of today's computer architectures, I have aimed to strike a balance and achieve the best possible design.

8.4

When working with a monolithic kernel like Linux, caution is necessary when adding new code and features. It's important to avoid creating new system interfaces as they become difficult to change once they are used. Other code additions, such as disk drivers, are less problematic as they don't introduce interface constraints.

Balancing the implementation and usefulness of a feature is crucial. Poorly designed interfaces or implementations can limit future developments and cause compatibility issues. For example, rigid filename length limitations in an interface can hinder future extensions. Microsoft's DOS/Windows file system interface is cited as an example of a bad interface.

The Plan 9 operating system's process fork system call is mentioned as another case where a clever feat ended up causing performance issues. Linux, on the other hand, implemented similar functionality properly. The importance of managing the development of Linux and making wise design decisions is emphasized, highlighting the need to keep the kernel small and minimize constraints on future development.

While Linux inherited some problematic interfaces from previous Unix implementations, maintaining compatibility with Unix applications has been crucial for its popularity. Despite some drawbacks, Linux strives to be as clean as possible while still providing compatibility.

8.5

Unix has been highly successful in terms of portability, thanks in large part to the widespread availability of C compilers on various architectures. The Unix kernel, like Linux, relies on C to achieve its portability. The importance of compilers led me to license Linux under the GNU Public License (GPL), which is also the license for the GCC compiler. While I consider other projects from the GNU group relatively insignificant compared to GCC, compilers are fundamentally essential. With the Linux kernel following a generally portable design, portability becomes feasible with the availability of a good compiler. Concerns about portability now revolve more around compilers than architectural differences. For instance, Intel's 64-bit chip, the Merced, poses challenges for compilers. Thus, the portability of Linux is closely linked to the fact that GCC is ported to major chip architectures.

8.6

Modularity is crucial for the open-source development model of the Linux kernel. It allows parallel work without clashes and enables easy integration of new features. With modularity, patches for new components can be added without impacting existing functionality. Loadable kernel modules were introduced in the 2.0 kernel, improving modularity by providing a structured approach to writing modules. It allowed coordination of developers and maintained control over the kernel itself. However, runtime loading of modules introduced challenges, especially regarding the GPL license and the definition of derived works. System calls were determined not to be linking against the kernel, ensuring that programs running on Linux are not covered by the GPL. This decision allowed commercial vendors to write proprietary programs without GPL concerns. The rules for module makers remained somewhat ambiguous, leaving room for potential misuse. However, the Linux community values the open-source development model, and attempts to create proprietary versions would undermine its appeal.

8.7

Linux has achieved design goals that were traditionally associated with microkernel architectures. By adopting a general kernel model based on common elements across architectures, Linux attains portability benefits without the performance drawbacks of microkernels. The use of kernel modules allows hardware-specific code to be isolated, ensuring a highly portable core kernel. This strikes a balance between putting all hardware specifics in the core kernel, which sacrifices portability, and placing them in user space, leading to slower and less stable systems.

The approach to portability in Linux has also benefited its development community. The decisions made for portability enable a large group of developers to work simultaneously on different parts of Linux without losing control over the kernel. The architecture generalizations provide a reference point for evaluating kernel changes and eliminate the need for separate code forks for each architecture. Furthermore, kernel modules offer a clear mechanism for independent work on specific components of the system.

8.8

Linux has reached a stage where major updates to the kernel are unlikely. The focus now is on supporting a wider range of systems and interfaces, such as clustering and supercomputing, laptops, and embedded systems. The embedded systems market will benefit from Linux's portability and the decreasing cost of powerful hardware. Symmetric Multi-Processing (SMP) will continue to be developed, with support for more processors. However, for extreme cases like sixty-four processors, a special version of the kernel may be required to avoid performance decreases for regular users.

Specific application areas, like web serving, will continue to drive kernel development, although it's important to remember that web serving is just one application among many. Optimizations in web serving have been limited by network bandwidth constraints, but future developments in faster networking may enable more intriguing possibilities.

The most exciting developments for Linux are expected to occur in user space rather than the kernel space. The evolution of user space applications, distributions like Red Hat, and tools like Wine (Windows emulator) will be more significant than kernel advancements. In the future, someone may come along with a leaner and more modern system, building on Linux's code and interfaces. The ability to provide binary compatibility and utilize existing code will be a testament to Linux's success.

Giving It Away

9.1

In the early days of Linux (1993), Linux CDs became popular, and we were intrigued by the development. We initially believed it was a fluke, but as Linux continued to improve and gain more users, we realized there was a solid economic model behind it. This led us to study the OS development and engage with key developers and users. Convinced of its potential, we established Red Hat Software in 1995. Our role at Red Hat is to assemble various software packages into a functional operating system, providing support and services to users. Our unique business plan focuses on delivering freely-redistributable software and empowering users with control over their operating system.

9.2

Making money selling proprietary binary-only software is not necessarily easier than with free software. Both types of ventures face challenges, and the success rate for software ventures, regardless of their nature, is generally low. The IP model of software development and marketing is difficult, similar to panning for gold during the gold rushes. However, when software companies succeed, they can generate significant profits, attracting people to take the risks associated with the industry.

While making money with free software is challenging, it is not inherently more difficult than with proprietary software. The key to success lies in building a great product, employing effective marketing strategies, prioritizing customer satisfaction, and developing a strong brand known for quality and customer service.

In the highly competitive software market, offering unique solutions to customers is crucial. Open Source software, with its stability, flexibility, and customization options, can be a competitive advantage. Leveraging the benefits of open-source software and devising effective monetization strategies are essential for companies in this space.

Many companies adopt a partially open-source approach, allowing free distribution for non-commercial use but requiring license fees for commercial use. However, it's important to note that we are still in the early stages of free software deployment and market share growth. If a company is not currently making money, it could be due to the relatively small market size compared to the vast number of DOS/Windows users.

9.3

Red Hat does not operate in the software business based on intellectual property ownership like most software companies. Instead, they adopted an economic model inspired by industries where basic ingredients or components are freely available. For example, in the legal industry, legal arguments become public domain after being used in a case. In the automotive industry, cars are assembled from various parts supplied by different manufacturers. Similarly, commodity industries rely on brand management to build strong brands associated with quality and reliability.

Red Hat saw an opportunity to offer convenience, quality, and define what an operating system can be in the minds of their customers. By consistently supplying and supporting a high-quality Linux OS, they aimed to establish a brand that Linux users prefer. They also recognized the importance of creating more Linux users while ensuring that those users choose Red Hat.

Drawing parallels with industries like bottled water, where consumers may choose a specific brand due to irrational fears or preferences, Red Hat understood that market size is vital. They benefit when other Linux suppliers successfully build demand for the product, as a larger overall user base increases the potential customer pool for Red Hat. The power of brands is significant in the technology business, as evidenced by venture capital investments in Open Source software companies that have established strong brand recognition and are known for their quality products.

9.4

Linux, as an open-source operating system, offers a unique market positioning compared to proprietary binary-only OSes. The primary complaint about the dominant market leader is their control over the industry. Linux, on the other hand, provides users with control over the OS platform, avoiding the pitfalls of becoming another proprietary OS.

Red Hat, as an OS assembly plant, selects the best open-source components to build a high-quality Linux OS. However, the control over the OS lies with the users, allowing them to make their own choices and modifications. This contrasts with proprietary OS vendors who control access to their source code and can charge tolls for using their APIs.

The appeal of open-source OSes lies in the control and freedom they offer to users. This model also benefits independent software vendors (ISVs), as they can sell their applications without worrying about the OS vendor being their biggest competitive threat. Companies like Corel, Oracle, and IBM have recognized the potential of Linux and have embraced it.

By offering users control over the technology they use, Linux provides a compelling advantage over proprietary binary-only OSes. While the existing OS vendors may eventually react to this model, the industry as a whole would benefit from better products at lower costs. Red Hat's goal is to continue growing and succeeding by carefully managing the Red Hat Linux brand and positioning Linux among OS alternatives.

Fermilab's experience highlights the importance of the control benefit of Linux. They chose Red Hat Linux because it provided the customization and performance they needed for their research projects. The Linux OS offers benefits and overcomes the limitations of proprietary binary-only OSes, making it an attractive choice for large organizations and technology suppliers.

Through brand management and market positioning, Red Hat enjoys growth and success, leveraging the benefits and market appeal of Linux in the competitive OS market.

9.5

The distinctive benefit of using Linux is the control it offers through its complete source code and the freedom to modify it without permission. This control is highly valued by organizations like NASA, where perfect reliability is crucial and access to source code is necessary to meet specific requirements. Unlike proprietary binary-only OSes, Linux allows users to customize the product to suit their applications.

Red Hat's unique value proposition lies in providing customers with this ability to modify and control their Linux OS. This proposition challenges traditional notions of intellectual property by emphasizing access and control over source code. Red Hat believes that the General Public License (GPL) from the Free Software Foundation aligns with the spirit of Open Source and is effective for managing cooperative development projects. The GPL ensures that modifications and improvements to the OS remain public, fostering collaboration among different teams.

Linux's modular nature, with over 435 separate packages, also influences licensing considerations. Red Hat needs a license that allows them to ship the software and make modifications to meet user needs. A less restrictive license requiring permission from multiple authors or development teams would be impractical. Red Hat prioritizes licenses that provide control over the software they use, enabling them to deliver the benefit of control to their customers, whether they are NASA engineers or application programmers.

While Red Hat values licenses that support their goals, they are not ideological about licenses and are open to options that grant control over the software. The focus remains on empowering customers and users with the freedom to customize and control their Linux experience.

9.6

The development of Linux challenges the stereotype of hobbyist hackers working in isolation. While individual contributors, including hobbyist programmers, play a valuable role, the majority of code in the Linux OS is created by professional software developers from major organizations. Linus Torvalds, the creator of Linux, started its development as a student, but many contributions come from professionals.

Examples of contributions from established organizations include the GNU C and C++ compilers from Cygnus Solutions, the X Window System from the X Consortium (with support from IBM, HP, Digital, and Sun), and ethernet drivers from NASA engineers. Device drivers are increasingly provided by device manufacturers themselves. The talent behind open source software is often the same as that behind conventional software.

An example illustrating the cooperative nature of open source development is Grant Guenther's work on developing a Zip drive driver for Linux. Grant, a member of Empress Software's database development team, needed secure file transfer between the office and home. Instead of purchasing an expensive proprietary solution, he chose to write a driver for Linux. He collaborated with Zip drive users across the internet to test and refine the driver. Grant's contribution demonstrates the cost-saving and mutually beneficial nature of open source development.

Cooperative models like the Open Source development model offer win-win propositions, allowing individuals and organizations to contribute and benefit from shared solutions, reducing costs and fostering innovation.

9.7

The Open Source model, including Linux, offers a unique benefit: control over software. Features such as source code access and Free Software licenses are important, but they are not the main reason for the growing adoption of Linux. In technical markets, the best technology does not always guarantee success. Linux's success hinges on its market positioning as more than "just another OS." It represents a revolutionary approach to software development that improves computing systems. The true benefit lies in the control it provides to companies over their software. Open Source is the best way to achieve this benefit, as it enables companies to have control and influence over their software, making it a compelling choice.

9.8

The Linux model stands in stark contrast to the fragmentation seen in the Unix ecosystem. While there are numerous incompatible versions of Unix, the forces at play in Linux are working towards unification. The primary difference lies in the proprietary nature of Unix, where vendors have incentives to keep their innovations exclusive. As a result, the various Unix versions diverge substantially over time. In Linux, the dynamics are reversed. When one Linux vendor adopts an innovation, others quickly follow suit because they have access to the source code and the freedom to use it. This collaborative approach fosters convergence around open standards and removes intellectual property barriers. An example of this is the adoption of newer glibc libraries by popular Linux distributions, which demonstrates the power of open source in driving unification and improvement.

9.9

In the face of revolutionary new practices, there are always skeptics who predict its downfall and ideologues who insist on a pure implementation. However, there are also those who embrace the new model and continue to innovate and test it in applications where it proves superior to the old one.

The primary benefit of this new technology model can be seen in the birth of the PC. When IBM introduced the PC in 1981, it wasn't because it was a better mousetrap. The early PCs had limited memory and basic features. However, what drove the PC revolution was the control it gave users over their computing platform. Users could choose different manufacturers, expand their memory, and select peripherals for specific purposes. Despite the inconsistencies and complexities that came with this model, consumers loved the choice and control it offered.

The PC hardware business did not fragment because specifications remained open and there was pressure to conform to standards for interoperability. No single vendor had a significantly better mousetrap that would lock users into a proprietary system. Innovations accrued to the community at large.

The Linux OS follows a similar path by providing consumers with choice over the technology that comes with their computers at the operating system level. While it requires a new level of responsibility and expertise from users, once they have experienced the choice and freedom of the new model, they are unlikely to return to the old model of being locked into a proprietary binary-only OS. Critics may find occasional problems with Linux, but consumers value choice, and the open-source software development marketplace will find ways to address and solve those issues.

Diligence, Patience, and Humility

The person who wrote this chapter smoked better weed than the others. He probably added special mushrooms as well. After reading the whole chapter I can confirm that it would have been better to drink a bottle of vodka, i wouldn't have wasted half an hour of my life and I would have come up with better ideas than this guy.

Open Source as a Business Strategy

11.1

Open-source software development has its tradeoffs, and its benefits may not apply to every situation. Analyzing long-term goals and competitive advantages is important before deciding on the development model to adopt.

Platforms, such as Win32 for Windows applications or CGI for web servers, define the software development process. They provide a set of rules and tools for developers to build upon. Platforms enable software compatibility and portability. While Win32 is specific to Windows, CGI is more portable and was implemented by major web servers for compatibility. Platforms are crucial for software development in both the Internet and computer environments.

In the Apache project, we implemented a powerful internal API that separated core server functionality from higher-level features. This allowed us to delegate the development of specific modules to dedicated teams, reducing the burden on the core development group.

Owning software platforms can be a profitable business model, but it limits competition and technological evolution. Competitors may have superior technology or services but are unable to utilize them due to restricted platform access. Customers can become dependent on a platform and face high switching costs when prices rise. To maintain freedom of choice and affordability, businesses should demand non-proprietary software based on open platforms.

The scalability of software production is different from traditional economics. Open-source implementations of protocols and APIs are crucial for long-term platform health. Commercial implementations can be acquired by competitors, undermining the independence of standards. Open-source software serves as a reference and facilitates comparison between different implementations.

Organizations like the IETF and the W3C play a role in standards development but cannot ensure widespread implementation. Demonstrating correct implementations through open-source work is often necessary.

AOL experienced compatibility issues with Apache, and we provided an unofficial patch to address the problem temporarily. Similar interoperability issues with other vendors' HTTP products have occurred, where vendors had to choose between fixing their bugs or abandoning sites that would be affected. Without an open-source reference web server like Apache, these subtle incompatibilities could have escalated, leading to separate webs built on different vendors' servers and clients.

Such fragmentation would have been a disaster for content providers, service providers, and software developers who rely on HTTP communication. The conflicting pressures to cooperate and innovate would have prevented standardization efforts. The case of client-side JavaScript exemplified this problem until the W3C intervened and established the Document Object Model (DOM) as a multiparty standard.

Closed software implementations, even with accidental misinterpretations of specifications, can lead to deviations and hinder stability. Building services and products on a standards-based platform ensures the stability of business processes. The success of the Internet highlights the importance of common platforms in facilitating communication and shifting focus towards creating value in the content exchanged rather than extracting value from the network itself.

11.2

Consider the revenue distribution of your products and services. If a significant portion of your revenue comes from a specific platform, giving it away for free may seem counterintuitive. However, by making the platform free, you can attract a larger user base and increase revenue from complementary services. Lower development costs and a competitive advantage against rivals are additional benefits. It's important to evaluate the market need and potential costs before implementing an open-source approach.

11.3

Launching an open-source project requires careful consideration. It's important to conduct a competitive analysis, assess the product's components, and determine the potential for success. Different strategies can be adopted, such as bundling certain functionalities, contributing to existing open-source projects, or combining open-source and commercial approaches. Analyzing market demand and engaging with potential contributors are crucial for project viability. Setting realistic goals and avoiding overreliance on dominating the market from the start is advisable.

11.4

When considering which parts of your product to open-source, it can be helpful to visualize a spectrum from infrastructural software to end-user applications. Open-source software has traditionally focused more on the infrastructural side due to factors such as the complexity of graphical interfaces and cultural preferences. However, there are exceptions, such as the GIMP, which combines features of both infrastructure and end-user application. By comparing your offering to competitors and drawing a vertical line, you can determine what to open-source and what to keep proprietary, establishing your platform. This line represents the interface between public code and private code, driving demand for your proprietary offerings.

11.5

When there is a commercial gap between two open-source software components, there is a natural pressure to bridge that gap with a public solution. If your company attempts to fill that gap with proprietary software, motivated developers may come together to create an open-source alternative, eliminating the advantage of your commercial offering. Relying solely on proprietary source code for revenue can be risky. A more solid strategy is to offer commercial additions or services that complement existing open-source offerings.

11.6

Open-source software is present in various standard software categories, particularly on the server side. While not every category has dominant open-source alternatives, there are usually decent open-source options available. Contributing code or enhancements to existing open-source projects can be a compelling strategy, provided that the licensing terms align with your goals and you can collaborate effectively with the existing developers. Developer satisfaction and collaboration are key to the success of open-source projects. While competition can be healthy, it is important to avoid unnecessary duplication of efforts. In certain cases, radical departures from existing solutions may be necessary to introduce innovation and address emerging challenges.

11.7

To ensure the health and evolution of an open-source project, it's crucial to have sufficient momentum and active developers. Building a core set of developers who are interested and invested in the project is essential. Depending on the complexity of the project, several roles and skills are necessary:

Infrastructure support: Someone to set up and maintain mailing lists, web servers, code repositories, bug databases, etc. Code "captain": Responsible for overseeing code quality, integrating contributions from third parties, and fixing bugs. Bug database maintenance: Maintaining an organized system for receiving and addressing bug reports and issues. Documentation/web content maintenance: Ensuring that non-technical users can understand and appreciate the software, providing architecture and procedural documentation. Cheerleader/evangelist/strategist: Building momentum, attracting developers and potential customers, and understanding the project's role in a larger context. These roles represent the minimum resources for a moderately complex project. While some roles can be shared or handled by groups, it's crucial to have dedicated resources, especially in the early stages. Additionally, these roles focus on maintenance rather than new development efforts.

If sufficient resources cannot be found to cover these roles and support basic new development, it may be necessary to reconsider open-sourcing the project.

11.8

Determining the appropriate license for your open-source project involves careful consideration and legal analysis. Here are three common types of licenses and their business considerations:

BSD-Style Copyright: This type of license, used by Apache and BSD-based operating systems, allows for maximum flexibility. It permits users to do what they want with the code, with the only requirement being proper attribution. From a business perspective, this license is beneficial when joining an existing project, as there are no restrictions on future use or redistribution. It allows for a mix of proprietary and open-source code, giving you more options. However, it doesn't incentivize companies to contribute their code back to the project, which may result in missed opportunities for collaboration.

Mozilla Public License (MPL): Developed by the Netscape Mozilla team, the MPL addresses some issues not covered by the BSD or GNU licenses. It mandates that changes to the distribution must be released under the same copyright, making them available back to the project. It also includes provisions protecting against patent issues in contributed code. While this is a positive aspect, there is a flaw in the MPL (as of December 1998) related to patent claims on the entire Mozilla codebase. This flaw may cause concerns for companies with significant patent portfolios. Despite this flaw, the MPL is a solid license that balances the flow of bug fixes and enhancements to the project while allowing for value-added features by commercial entities.

Choosing the right license depends on the nature of your project. The BSD license is suitable for "invisible" projects like operating systems or web servers, while the MPL is more suitable for end-user applications where patents may be a concern and branching the project is more likely.

Ultimately, it's recommended to consult with legal experts to ensure compliance with licensing requirements and to align with your business goals.

The GNU Public License (GPL) has certain aspects that can be attractive for commercial purposes, despite not being initially perceived as business-friendly. The GPL mandates that any enhancements, derivatives, or code that incorporates GPL'd code must also be released as source code under the GPL. This viral behavior ensures that code remains free and prevents commercial interests from forking their own development version without making it public. While this may seem restrictive from a commercial perspective, the GPL can be effectively used to establish a platform that discourages competitive platforms and protects your position as the premier provider of products and services built on that platform.

One example is Cygnus and GCC. Cygnus profits by porting GCC to different hardware platforms and maintaining those ports. Most of their work, in compliance with the GPL, is contributed back to the GCC distribution and made available for free. Cygnus charges for the porting and maintenance efforts, not for the code itself. This strategy ensures that competitors cannot exploit a commercial technical niche on top of the GCC framework without giving Cygnus the same opportunity to do so.

Another way to use the GPL for business purposes is by offering a non-GPL'd version of the code for a price. For instance, if you have a program for encrypting TCP/IP connections, you can release it under the GPL, making it free for non-commercial use, but requiring commercial users to make their entire product GPL'd. However, you can maintain a separate branch of the code that is not under the GPL and commercially license it. This allows you to generate revenue from users who want to embed or redistribute the code for profit but are not willing to make their entire product GPL'd.

It's important to ensure that any contributions from third parties explicitly permit their inclusion in the non-GPL version. By treating contributors well and potentially compensating them for their contributions, this model can be successful. Companies like Transvirtual in Berkeley have applied this model to commercial projects, such as a lightweight Java virtual machine and class library.

The open-source license space will continue to evolve as people explore what works and what doesn't. Remember that you have the freedom to create a new license that precisely describes your desired position on the licensing spectrum. Offering more freedoms to users and contributors will incentivize them to contribute to your project.

11.9

The Apache Project utilizes key tools for effective open-source development. Concurrent Versioning System (CVS) enables shared code repositories, facilitating collaborative coding and bug tracking. Discussion forums help developers and users engage in project-related discussions, while separate lists ensure focused communication. Bug tracking using GNU GNATS allows for efficient bug reporting, specialization, and easy searching. These tools play crucial roles in managing distributed development and maintaining code quality. Launching and managing a successful open-source project requires dedicated effort and strategic implementation.

The Open Source Definition

12.1

The concept of free software dates back to the early days of computers when software was freely shared among researchers. The idea gained popularity through Richard Stallman and the Free Software Foundation's GNU Project. Stallman's GNU General Public License (GPL) codified the rights he believed users should have, promoting the concept of copyleft. The Open Source Definition, influenced by Stallman's ideas, emerged from the Debian project's efforts to define what constituted free software. Eric Raymond's essay "The Cathedral and the Bazaar" inspired the formation of the Open Source Initiative (OSI) and the marketing of free software to a wider audience. The Open Source campaign faced criticism and debates, including tensions between Stallman and Raymond, but ultimately defined a set of guidelines to distinguish open source from proprietary software.

12.2

The case of KDE, Qt, and Troll Tech exemplifies the impact of the Open Source Definition on non-Open Source products within the Linux infrastructure. KDE, an attempt at a free graphical desktop for Linux, relied on the proprietary Qt library from Troll Tech. This led to objections from Open Source proponents who believed that partially-free items like Qt blurred the definition of free software. The KDE developers negotiated a KDE Free Qt Foundation agreement with Troll Tech, but a fully Open Source competitor called GNOME emerged. To address the resistance, Troll Tech eventually released a fully Open Source license for Qt, defusing the conflict. The GNOME project continued to compete with KDE, aiming for superior functionality. The growth of Open Source contributions and complete Open Source Linux distributions showcased the increasing momentum of the Open Source movement.

12.3

The Open Source Definition, version 1.0, is a set of criteria that a software license must meet to be considered open source. The criteria include:

Free Redistribution: The license should allow selling or giving away the software without additional fees. Source Code: The program must include source code and allow distribution in source code form. Derived Works: The license must permit modifications and derived works, allowing them to be distributed under the same terms as the original software. Integrity of the Author's Source Code: Source code modifications should be allowed but kept separate from the original author's work. No Discrimination Against Persons or Groups: The license should not discriminate against any person or group. No Discrimination Against Fields of Endeavor: The license should not restrict the use of the program in specific fields. Distribution of License: The rights of the license should apply automatically to all recipients without the need for additional licenses. License Must Not Be Specific to a Product: The rights granted by the license should not depend on the program being part of a particular software distribution. License Must Not Contaminate Other Software: The license should not impose restrictions on other software distributed alongside the licensed software. Example Licenses: The GNU GPL, BSD, X Consortium, Artistic, and MPL licenses are considered conformant to the Open Source Definition. Minor revisions may be made to the Open Source Definition in the future, but the intent of the document will remain largely unchanged. The mentioned licenses are examples of conformant licenses, but if any of them were to change, the Open Source Definition might need to be revised accordingly.

12.4

The section discusses common licensing practices related to open source software. It clarifies the misconception that free software is often considered public domain when, in fact, it is copyrighted and covered by a license. Public domain software, on the other hand, is intentionally released without copyright restrictions.

The section also emphasizes that open source software licenses, including the GNU General Public License (GPL) and the GNU Library General Public License (LGPL), do not grant ownership of the software to the user. Instead, they provide rights and permissions to use, modify, and distribute the software within the terms of the license.

The GPL, LGPL, and other open source licenses disclaim all warranties to protect the software authors from liability. If authors lose this protection, it could discourage them from contributing free software.

The section provides an overview of different licenses, including the GPL, LGPL, X license, BSD license, Apache license, Artistic license, Netscape Public License (NPL), and Mozilla Public License (MPL). It highlights the differences between these licenses, such as the ability to take modifications private, the requirements for mentioning the software's origin, and the ability to incorporate the software into proprietary programs.

While the GPL and LGPL are recommended licenses, the section acknowledges that some developers may choose alternative licenses based on their own reasons. However, it emphasizes the importance of carefully considering the implications of creating a new license and the potential long-term consequences for software users.

12.5

Avoid creating a new license if existing licenses are suitable: The proliferation of numerous incompatible licenses can be detrimental to open source software. It is recommended to use one of the established licenses rather than creating a new one.

Consider the ability to take modifications private: If you want to ensure that modifications made to your software are shared back with the community, licenses such as the GNU General Public License (GPL) or Lesser General Public License (LGPL) are good choices. If you are comfortable with people keeping modifications private, licenses like the X or Apache license can be used.

Allow merging with proprietary software: If you want to permit others to merge your program with their proprietary software, the LGPL explicitly allows this while still requiring modifications to the original code to be shared. Alternatively, licenses like the X or Apache license allow modifications to be kept private.

Dual-licensing for commercial versions: If you want to offer a commercial version of your program that is not open source, you can consider dual-licensing. The GPL can be used as the open source license, while a separate commercial license can be offered for those who wish to use the software without open source requirements.

Consider the payment model: If your primary goal is to have everyone using your program pay for it, open source may not be the most suitable approach. However, many open source authors view their programs as contributions to the public good and are not primarily concerned with monetary compensation.

It's important to note that these points are general considerations, and the choice of license depends on the specific goals and requirements of your project. It's recommended to carefully study and understand the terms and implications of any license before selecting one for your software.

12.6

IBM, Intel, Netscape, and other companies are embracing Open Source and investing in Linux-related projects. The leaked Halloween Documents reveal Microsoft's concern about the threat posed by Open Source and their potential strategies to counter it, including copyrighted interfaces and patents. The text also mentions the importance of defending against Trojan horses and the need for improvements in areas like user interfaces and system administration. Despite the challenges, the author predicts that Open Source will ultimately succeed due to its adoption by computer science students, research laboratories, and businesses.

Hardware, Software, and Infoware

The concept of a "killer application" has evolved in today's digital age. Instead of traditional desktop productivity tools, it is now found in individual websites. Take Amazon.com, for example. I recently spoke with friends who were considering buying a computer solely to access Amazon.com and make online purchases.

This highlights the transformative power of web-based applications, which I like to call "information applications" or "infoware."

Infoware automates tasks that were previously difficult to handle in the traditional computing model. For instance, searching a vast book database or buying a book from home can now be done effortlessly by anyone, without specialized training. Web-based applications use intuitive interfaces, using plain language and visual elements instead of complex controls found in traditional software.

Behind the scenes, these applications leverage scripts and the Common Gateway Interface (CGI) to interact with external programs and generate dynamic web pages. The rise of infoware has revolutionized decision-support applications and made information more accessible for users. Open-source technologies have played a crucial role in powering the web, including the TCP/IP network protocol, DNS, and scripting languages like Perl, Python, and Tcl. These technologies have enabled the growth of web-based dynamic content and contributed to the success of websites like Yahoo!

Amazon.com is not only a user of Perl but also showcases its power as a "glue language." The authoring environment at Amazon.com demonstrates how Perl integrates various computing tools. It serves as a bridge between Microsoft Word or GNU Emacs, CVS, and Amazon.com's SGML tools. Perl plays a crucial role in rendering different sections of the website and converting SGML into HTML for approval. Its versatility and lightweight nature make it ideal for quick and effective solutions in the ever-changing web environment.

Microsoft's attempt to transform infoware back into traditional software with ActiveX was unsuccessful. Paradigms in the computer industry often shift, and existing players have a vested interest in maintaining the status quo. IBM's loss of dominance to Microsoft is a prime example of this. Microsoft failed to recognize the shift from proprietary to commodity hardware and the rising importance of software in value creation.

The advent of the PC and open systems platforms like Unix disrupted the industry by lowering the barriers to entry. Entrepreneurs like Mitch Kapor and Bill Gates seized the opportunity. Similarly, open-source software has significantly reduced barriers to entry in the software market. It allows users to try new products for free, customize them, and contribute back to the community. The availability of source code for peer review and the freedom to experiment have fostered innovation and evolution in open-source projects, unconstrained by marketing barriers or bundling deals. The success of open-source software lies in its diversity and idiosyncratic nature. Perl, for example, started as a tool to automate system administration tasks and grew organically in unexpected directions. Its perceived chaos actually reveals rich structure and allows it to model complex problems effectively. Open source thrives on a tight feedback loop with customer demand, free from marketing influences and top-down decision-making. This bottom-up approach is ideal for solving real-world problems.

Entrepreneurs like Jerry Yang and David Filo leveraged open-source software and its simpler development paradigm to build successful companies like Yahoo!. The world's largest and most prosperous websites, including Yahoo! and Amazon.com, are built on open-source foundations such as FreeBSD, Apache, and Perl.

The next stage of the computer industry revolves around commoditizing the previous stage. Open source doesn't aim to defeat Microsoft in its own game but rather changes the nature of the game itself. Software's role will increasingly be as an enabler for infoware, and the focus will shift to information-application providers. Commercial opportunities exist in providing web servers, database backends, application servers, and network programming languages that align with the new paradigm.

The real challenge for open-source software lies in developing a business model that positions it as the "Intel Inside" of the next generation of computer applications. The pioneers of open source have already created a fork in the road, setting the stage for a radical reshaping of the computer industry landscape in the coming years.

Freeing the source

During the development of Netscape Navigator, the team referred to the browser's codebase as "Mozilla." The term originated from an inside joke but eventually became the official code name. As the open-source movement gained momentum, "Mozilla" became the generic term for web browsers derived from the Netscape Navigator source code. The mission was to "Free the Lizard," symbolizing the liberation of the code.

Preparing the code for open sourcing required extensive work. Issues were categorized, and the team tackled them at a rapid pace. One major challenge involved the disposition of third-party modules included in the browser. Over seventy-five modules required engagement with their respective code owners. Teams of engineers and evangelists were formed to persuade each company to join Netscape on the path to open source. Companies had to decide whether their code would be removed, replaced, shipped as a binary, or included as source code alongside Communicator.

Meeting the deadline for Project Source 331 was crucial, leading to tough decisions. Third-party developers were given an ultimatum: either be part of the project by February 24th or have their code removed. The process became increasingly challenging as the deadline approached. For instance, Java, a proprietary language, had to be removed from the codebase. Three engineers were assigned to perform the complex task of disentangling Java code from the browser.

Cleansing the code was an enormous undertaking. Despite initial doubts about meeting the deadline, strategies were devised, and the entire team focused on the task. Each third-party participant's involvement was resolved, and all comments were edited out of the code. Teams were assigned responsibility for specific modules and worked diligently to scrub the code. An innovative decision was made to utilize the Intranet bug-reporting system, called "Bugsplat," as a task manager. It served as an efficient workflow management system, tracking the progress of modules and facilitating communication among team members.

Another significant challenge was the removal of cryptographic modules due to government regulations. The engineering team had to not only remove cryptographic support but also redact all related hooks. One team was dedicated to maintaining constant communication with the NSA and managing compliance issues.

The process of open sourcing the Netscape Navigator code involved overcoming numerous obstacles, making tough choices, and leveraging innovative approaches to meet deadlines and ensure compliance with regulatory requirements.

14.2

During the open sourcing of the Netscape Navigator code, the team faced the challenge of finding a suitable license. They explored existing licenses such as the GNU General Public License (GPL), GNU Library General Public License (LGPL), and BSD license but found them inadequate for their unique needs. A group of Open Source community leaders, including Linus Torvalds, Eric Raymond, and Tim O'Reilly, provided insights and engaged in discussions to craft a new license.

After extensive research and discussions with the Netscape legal team, a new license called the Netscape Public License (NPL) was created. This license aimed to strike a balance between promoting free source development by commercial enterprises and protecting free source developers. It underwent beta-testing and received feedback from the community, leading to revisions and the release of a second license called the Mozilla Public License (MozPL), which was nearly identical to the NPL but with additional rights granted to Netscape.

The licenses outlined the conditions for using and modifying the Netscape code. All code initially released was covered by the NPL, and modifications had to be released under the same license. New code could be released under the MozPL or any compatible license. The licenses also addressed the distinction between bug fixes and new code, considering the concerns of developers.

The goal of releasing the source code under these licenses was to engage the wider community in innovating the browser market and to foster corporate interest in open-source development. Netscape wanted to create an environment where large corporations could adopt the open-source model and contribute to the movement.

The licensing effort was seen as a crucial part of the open-sourcing process, and the active participation of the community in shaping the licenses demonstrated the true spirit of open-source development. The team was motivated by the vision of talented programmers worldwide contributing to the codebase and infusing the browser with their creativity and innovation.

14.3

After Netscape announced its plan to release the source code, mozilla.org was established as the central hub for the open-source project. The organization was created to fulfill the role of maintainer, coordinating contributions from developers worldwide. Setting up funding, machines, and mailing lists, mozilla.org aimed to be ready for the release of the code. It was crucial for mozilla.org to be separate from Netscape's Product Development Group to ensure that the organization's goals were focused on coordinating the project rather than solely shipping Netscape products. The team recognized the need to have a dedicated entity managing the open-source project to collect and incorporate patches and updates from contributors. By establishing mozilla.org as the maintainer, they aimed to provide a central depot for the project and foster collaboration within the wider developer community.

14.4

As developers submit code changes to mozilla.org, the organization plays a crucial role in deciding which code is accepted. The primary considerations are the quality and merit of the code, as well as its compatibility with the Netscape Public License (NPL). Contributions under incompatible licenses are not accepted to avoid complexities and legal issues. Each major module in the Mozilla code base, such as the Image Library or XML Parser, has a designated owner who knows the code best. Module owners, including both Netscape engineers and external contributors, submit their modifications to mozilla.org for inclusion in distributions. In case of disagreements, mozilla.org acts as an arbitrator, making the final decision. To manage the collaboration between internal Netscape developers and external contributors, mozilla.org utilizes tools like Bonsai and Tinderbox. Bonsai allows developers to track code changes and detect issues, while Tinderbox provides visibility into the source tree, including successful builds, broken platforms, and file states, facilitating problem identification and resolution.

14.5

With the code release deadline approaching, there was a sense that a celebration was needed. Jamie proposed a groundbreaking idea to rent out a nightclub in San Francisco, invite the public, and broadcast the event over the internet. The idea of inviting non-employees to the party was initially met with surprise, but then embraced with enthusiasm. The "Mozilla Dot Party" took place on April 1st at The Sound Factory, one of the biggest nightclubs in San Francisco. The party attracted over 3,500 attendees, including influential figures in the software community. DJs, including Apache founder Brian Behlendorf, gave away mozilla.org T-shirts, software, and various items. Inside the nightclub, projection screens displayed scrolling lines of the Mozilla code, creating an immersive atmosphere. The celebration marked the liberation of the code and symbolized the beginning of a new era.

The Revenge of the Hackers

15.1

In late 1993, the author's first encounter with Linux through the Yggdrasil CD-ROM distribution was a shock to their assumptions about hacker culture. They had believed that amateur hackers couldn't produce a usable multitasking operating system, especially considering the struggles of the HURD developers. However, Linux and its community proved them wrong. Linux not only met the minimum requirements of stability and Unix interfaces but exceeded them by providing a vast array of programs, resources, and tools. This experience made the author feel more connected to the hacker community, as their own free-software projects were part of this vibrant ecosystem. Intrigued by Linux's success despite the complexities of software engineering, the author spent years studying and observing the Linux community to understand their approach. This eventually led to the writing of "The Cathedral and the Bazaar," where the author shares their observations and theories on how the Linux community achieved such high-quality results.

15.2

The author reflects on how they observed the hacker community evolve an effective software development method without being aware of it. They realized that the lack of a theoretical framework and language hindered their ability to systematically improve their methods and explain them to others. Therefore, the author wrote "The Cathedral and the Bazaar" (CatB) to provide the hacker culture with a language to explain itself. The paper did not introduce any new methods but presented existing practices through metaphors and a compelling narrative. When the author presented CatB at Linux Kongress and later at Tim O'Reilly's Perl Conference, it received enthusiastic responses. The speech gained attention, especially among individuals from Netscape Communications, Inc., which was facing challenges from Microsoft's dominance. CatB played a role in advocating for Netscape to open its browser source code, as it provided strong reasons to believe that openness could counter Internet Explorer's dominance.

15.3

After Netscape announced its decision to release the source code of the Netscape client line, the author learned that their work had been described by CEO Jim Barksdale as "fundamental inspiration" for the decision. This news came as a shock to the author, as it marked the first time a Fortune 500 company had embraced the hacker culture and its principles. Realizing the significance of this moment, the author felt a strong sense of responsibility to support Netscape's gamble and ensure its success. They recognized that if Netscape failed, it would discredit the hacker culture and impede progress for years to come. To help Netscape, the author offered their assistance in developing the license and strategizing the details of the project. They flew to Netscape headquarters for meetings and collaborated with key individuals in the Silicon Valley and national Linux community. It became evident that a longer-term strategy was needed to build upon the Netscape release, leading to the development of a plan to sustain and advance the open-source movement.

15.4

After the decision to embrace open source at Netscape, the strategy for promoting and advancing the movement became clear. The author and the open source community recognized the need for a marketing campaign to rebrand and build a positive reputation for open source software. They understood that the term "free software" had negative connotations and needed to be replaced with a more appealing and pragmatic term. Thus, the term "open source" was coined during a meeting in Mountain View.

The strategy involved top-down evangelism, targeting CEOs, CTOs, and CIOs directly to promote open source software. Linux, with its name recognition, broad software base, and large developer community, was identified as the best demonstration case for the open source movement. The Fortune 500 companies became the primary target market, given their concentration of resources and influence on the software industry.

To capture the attention of decision-makers and investors, it was essential to co-opt prestigious media outlets such as the New York Times, Wall Street Journal, and Forbes. Educating the hacker community in guerrilla marketing tactics was seen as crucial to ensure consistent messaging and effective arguments. Additionally, registering the term "open source" as a certification mark, tied to the Open Source Definition, was done to protect the term from potential misuse or dilution.

These strategic elements formed the foundation for promoting open source software and advancing its acceptance and adoption in the corporate world. The author and the open source community recognized the significance of Netscape's decision and the opportunity it presented to challenge the status quo and redefine the perception of hacker-driven innovation.

15.5

The author recognized the need for a charismatic spokesperson and media personality to effectively convey the open source movement's message to the press and the public. They understood that the press is more likely to pay attention when ideas are presented by larger-than-life personalities and through engaging stories, drama, and sound bites. As an extroverted individual with experience dealing with the press, the author felt they were well-suited for this role.

However, they also recognized the personal sacrifices involved. Becoming a public figure and media personality meant sacrificing privacy and potentially facing criticism and backlash from both the mainstream press and their own hacker community. Additionally, taking on this role would require a significant amount of time and dedication, possibly taking them away from their hacking activities.

Despite the challenges, the author made the decision to embrace the role of an evangelist, understanding that it was necessary to promote the open source movement and achieve success. They developed a theory of media manipulation, focusing on creating attractive dissonance to generate curiosity and effectively promote their ideas.

The combination of the "open source" label and the deliberate promotion of the author as an evangelist led to increased media coverage of Linux and the open-source world. While a significant portion of the coverage quoted the author directly or used them as a background source, there was also criticism from a minority of hackers who viewed the author as egotistical.

From the beginning, the author planned to eventually hand off the evangelist role to a successor, whether an individual or an organization. They believed that, over time, institutional respectability would become more important than personal charisma. Currently, the author is working on transferring their connections and reputation to the Open Source Initiative, an incorporated nonprofit organization specifically created to manage the Open Source trademark. Although they are currently the president of the organization, they hope to eventually step down from that position.

15.6

After the adoption of the term "open source" at the Free Software Summit on March 7, the open-source movement gained momentum within the hacker community. By six weeks after the Mountain View meeting, a majority of the community had embraced the "open source" label and its associated arguments.

In April, the focus shifted to recruiting open-source early adopters to make Netscape's move appear less singular and to provide insurance in case of any setbacks. Although Linux was progressing technically and gaining positive coverage in the trade press and mainstream media, there was still a sense of fragility. Community participation in Mozilla had slowed down, independent software vendors were hesitant to commit to Linux ports, and Netscape's browser was losing market share to Internet Explorer. A serious setback could lead to negative backlash.

The first significant breakthrough after the Netscape announcement came with Corel Computer's announcement of its Linux-based Netwinder network computer on May 7. However, it was the mid-July announcements by Oracle and Informix about Linux ports that provided the much-needed support from industry leaders, alleviating concerns and solidifying the movement.

From mid-July to early November, the focus shifted to consolidation. The open-source movement began to receive steady coverage from the elite media, with articles in The Economist and a cover story in Forbes. Hardware and software vendors started showing interest in the open-source community and developing strategies to leverage the new model. Additionally, Microsoft's worries about the open-source movement became apparent when the "Halloween Documents" leaked. These documents, which showcased the strengths of open-source development and confirmed suspicions about Microsoft's tactics, received significant press coverage and further fueled interest in the open-source phenomenon.

The release of the Halloween Documents led to a request for the author to conference with major investors at Merrill Lynch on the state of the software industry and the prospects for open source. This marked an important turning point as Wall Street began to show interest in the open-source movement.

15.7

During the ten months following the Netscape release, Linux made significant strides in its technical capabilities. It developed solid SMP support and completed the 64-bit cleanup, laying the groundwork for future advancements. Linux's success in the film industry, such as rendering scenes for the movie Titanic, impressed graphics engine builders. The Beowulf project demonstrated the applicability of Linux's collaborative approach to high-performance scientific computing.

In the market, Linux continued to gain momentum while proprietary Unix systems lost market share. By mid-year, only Windows NT and Linux were gaining market share in the Fortune 500, with Linux gaining at a faster pace. Apache maintained its lead in the web server market, and Netscape's browser started making gains against Internet Explorer after a period of decline.

These technical and market developments further solidified the perception of Linux and open-source software as viable alternatives, increasing their appeal and acceptance in various industries.ù

15.8

The passage highlights the author's predictions for the future of open-source software, specifically Linux, in the near term (one year) and medium term (eighteen to thirty-two months).

In the near term:

The open-source developer community will continue to grow rapidly due to the affordability of PC hardware and internet connections. Linux will maintain its leading position, driven by its large developer community and widespread industry support. Increased commitments from independent software vendors (ISVs) to support the Linux platform, exemplified by Corel's commitment to ship their office suite on Linux. The Open Source campaign will raise awareness and gain traction among CEOs, CTOs, CIOs, and investors, influencing MIS directors to consider open-source products. Samba-over-Linux deployments will replace more NT machines, even in organizations with all-Microsoft policies. Proprietary Unix systems will continue to lose market share, with the possibility of one weaker competitor folding. In the medium term:

Support operations for commercial customers of open-source operating systems will become a significant business. Open-source operating systems, particularly Linux, will dominate the ISP and business data-center markets, while NT struggles to compete. The proprietary-Unix sector will collapse, with Solaris potentially surviving on high-end Sun hardware. Windows 2000 may face significant challenges or be unsuccessful, impacting Microsoft strategically but not immediately affecting its desktop market dominance. The future beyond two years becomes more uncertain, with factors like the potential breakup of Microsoft, emergence of new open-source or closed-source OSes, and global economic conditions playing a role. The crucial challenge for the Linux community will be delivering a user-friendly GUI interface that can rival Microsoft's dominance on the desktop. Ergonomic design and interface psychology pose challenges that hackers historically struggle with. The passage concludes by emphasizing the importance of designing software and hardware for all users, not just technical experts. The goal is to reduce complexity and create a user-friendly experience comparable to the Macintosh while retaining the strengths of the traditional Unix approach.

In summary, the author envisions Linux's continued growth and dominance in server, data center, and ISP markets, while Microsoft maintains its grip on the desktop. The ultimate challenge lies in developing a user-friendly interface that appeals to a broader audience and serves the needs of all users.