OPENING THE CODE: SOFTWARE EXCELLENCE AS A FUNCTION OF ITS DEVELOPMENT ENVIRONMENT

A Thesis submitted to the

Faculty of the Graduate School of Arts and Sciences

of Georgetown University

in partial fulfillment of the requirements for the

degree of Master of Arts in Communication, Culture, and Technology

by

David S. McLaughlin, B.S.

Washington, DC

1 May 2001

Abstract

Software is an increasingly important component of the modern world; indeed, software forms the architecture around which today's digital society is built. Because of the complex interrelationship between computer code, liberty, and the distribution of power in society, the environment in which code is produced makes a tremendous difference in the resulting structure of the world. An "Open" foundation for software development provides a variety of economic and societal benefits, but there are a number of preconditions necessary for its full deployment. One of the primary stumbling blocks is the need for the Open development process to demonstrate its capacity for producing "excellent" software. This thesis tackles that obstacle by first defining excellence in software, and then examining the software ecosystems that surround proprietary and Open Source conceptions of digital property in terms of their incentive structures—incentives that either encourage or discourage quality. Herein, this thesis demonstrates that not only can an Open Source development model produce quality software, but it also takes advantage of the complex adaptive systems model of diversity and adaptation to better meet the established criteria of an "ideal" software development system.

Acknowledgments

I would like to thank my parents for their love and support throughout all the years of my education, including my time at Georgetown University. In addition, I would like to thank my thesis advisor Linda Garcia for her encouragement in the development of this project.

Contents

1 Introduction: The Complex World of Software Development

"Use the Source, Luke."
—With Apologies to Star Wars[1]

The software development process, particularly of large-scale projects and applications, is an example of a complex system.[2] Software development is not a linear function with a single set of dependent and independent variables. Software development inhabits a world of many different types of players, each of which interact in a variety of unexpected ways that constantly modify the entire world's direction.[3] Even small changes in one particular condition may drastically alter the outcome, such as the oft-mentioned example of a butterfly's flapping its wings in Kenya leading to a hurricane in Alabama. In a sense, software development is more of a garden to be cultivated than it is an engineering science to be systematically delineated.

[1]For a series of comics that plays on the theme of Open Source as the Rebel Alliance in Star Wars, as well as highlights the ideological motives present in many members of the Open Source community, please see the online strip *User Friendly* beginning with the sequence at http://ars.userfriendly.org/cartoons/?id=19981127.

[2]For an introduction to the world of chaos theory and complex systems, please refer to [1].

[3]Please see [2].

As a complex adaptive system, software development comprises many ingredients that all must enter the mix for a successful, innovative, and widely-accepted set of code. Depending on the initial conditions established in the foundation for its development, software can grow and evolve in a variety of wildly different ecosystems. In particular, varying conceptions of the property rights surrounding the nature of the digital bits and algorithms at the core of the software industry lead to differing scenarios of evolution. Whether software is treated as "Closed" or "Open" is a critical variable in an overall development environment. Changing this axiom regarding usage "rights" of the code influences the overall incentive structure for the production and use of the resulting software. Thus, whether code is "Closed" or "Open" determines two distinct ecosystems with varying structures.

In spite of these varying environments, software today still can be evaluated in terms of its merits and its success, both technically and in the market. Because software is instrumental in shaping our entire networked society, such evaluation is absolutely necessary. To function well technically and to thrive in the marketplace, the "ideal" software development ecosystem must be able to meet a variety of criteria, more fully discussed in chapter 2.

First, the structure constructed on the ecosystem's particular axioms must be able to support innovation and development and encourage creative solutions. In addition, the software development ecosystem must be able to support itself economically within the overall framework of today's market driven economy of global capitalism. The software that this framework produces must be stable, powerful, secure, and accessible to both users and developers in order to expand out of a niche role; thus, the interface must be clear and available publicly for standardization to proceed. Finally, the software and its development ecosystem must successfully leverage network effects and garner the necessary support and marketing within the business community to expand into a position of market leadership. Both "Closed" and "Open" development environments can be thus evaluated in terms of their fulfilling these criteria for software excellence.

1.1 Closed and Open Development Environments

Since its inception, software, and digital information in general, has exhibited a tension with respect to varying conceptions of property rights—control over the bits. Software entails a high initial cost of development, but an extremely small cost of replication (high initial costs, and low marginal costs).

Given these conditions, software benefits greatly from economies of scale, and is highly susceptible to network effects. Thus, the first mover in a particular area gains a tremendous advantage over competitors through sheer momentum and the network externalities associated with existing connections.

Because early successful commercial forays thrived under a paradigm of digital property rights which emphasized "selling the bits," the software market today is dominated by proprietary software, controlled and published by single organizations that obtain much of their revenue streams from the licensing and distribution of their product. Nowhere is this model more apparent than in the desktop market, where a single corporation—Microsoft—controlled 87% of sales for client computer operating systems in 1999.[4] The software and computing market, a prime example of networking effects at work, has consolidated around a set of products owned and controlled through traditional intellectual property structures.

There is, however, a competing paradigm of property rights for digital products—one that is based on the free sharing of information and software as its basis, and that formulates a different foundational structure for rules

[4]Please see [3].

negotiating ownership and control. This "new" movement, embodied in the Open Source community and the free software movement, suggests that a development ecosystem based on the fundamental concept of freely-shared information, combined with massive, network-enabled collaboration without centralized control, has the potential to generate innovative and quality software products. The emphasis in this model of software development is not on the "freeness" of the resulting software in terms of costs, but rather on the "freeness" of the software in terms of its openness to examination, its extensibility, and its position as a catalyst for creativity. In the words of Chris DiBona, "English handles the distinction here poorly, but it is the distinction between gratis and liberty, as in 'Free as in speech, not as in beer.'" The Open Source movement is a competing paradigm to the proprietary software model based on a completely different set of core axioms regarding control of the digital bits.

1.2 Quality Software: Development Ecosystems Matter

The Open Source paradigm, competing with the entrenched ecosystem of proprietary software development, has the potential to disrupt the current power holders not only in the desktop computing market, but also in the en-

tire software industry. Thus, there is a tension between the two environments in which software development can occur. On one hand, current market leaders emphasize development of software based on tightly controlled property rights. On the other, Open Source proponents claim that a different scheme of control over code allowing for greater freedom will lead to better, more innovative software which is more advantageous to all.

The question then arises, can an ecosystem of software development, based on the free exchange of its core assets, produce software the quality of which is equal to or greater than that produced under the current development model, which emphasizes proprietary control and licensed distribution?

This paper will answer the question of whether software development based on an open foundation produces results that not only are closer to the "ideal," but also are well-suited to overcome the network effects of entrenched proprietary development ecosystems. To test this hypothesis, components of an "ideal" software development ecosystem will first be constructed based on technical considerations of software design and software project organization, evolution, and management. In the context of this theoretical idealized

ecosystem, the history and development of the proprietary and open models for software development will be examined in turn to see how they fit into the structure established for successful software production.

The tension between proprietary and Open Source development ecosystems will then be examined in detail through a case study of a particular software niche: the client desktop graphical operating environment. A case study methodology is particularly useful because it allows one to bring in multiple variables and unpack their subtle interrelationships. A least-likely and critical case study will be examined—that of the client desktop graphical operating system. This particular case is one of the most difficult in software. A successful, quality desktop environment relies on a tremendous amount of interacting components, all of which must be coordinated successfully to provide a solid total experience. In addition, the desktop is critical; if Open Source will succeed in other areas, it must be able to succeed in the area of a working user environment. In this particular software niche, there are powerful proprietary incumbents that control an overwhelming percentage of the market. The incumbents, as well as the Open contenders, will be examined in the context of the theoretical model. If software development

based on the Open Source paradigm can succeed in this case, the model should be replicable in other areas as well.

This thesis posits that not only will the ecosystem of Open Source development meet the criteria of an "ideal" software development system; it will meet these criteria better than a proprietary-based paradigm due to the evolutionary variation model at its heart. While this thesis focuses on the impact of Open vs. Closed development models for software in terms of "quality", the complex nature of the software development world leads to other implications as well. Open vs. Closed models of software development lead to differing architectures at the heart of the digital world, and architectures lead to differing systems of control.[5] Once this thesis has established the ability of an Open Source development environment to produce quality software, it will touch briefly on Open Source's potential for success in greater market acceptance. Because it is based on the sharing of its core assets, the Open Source environment of development will be able to effect change in the networks-effect laden market of the software industry. If the ecosystem of Open Source development is successful in the software industry, such a

[5] "Code is law." Please see [4].

revolution in the nature of property rights at the core of today's information economy could have long-lasting and far-reaching consequences for all sectors that rely on the creative production and dissemination of information-based products.

2 Software Excellence: Measuring Quality at the Heart of Today's Digital World

At the core of today's information-based economy lies software—the instructions that control the millions of processors, switches, and computers that aggregate, analyze, and manipulate data vital to the smooth functioning of processes around the world. Today's world is awash in information, and managing this massive torrent requires sophisticated algorithms and complex processing systems. Software is critical in maintaining these processes both on world-wide and personal levels.

Computers are processing devices that manipulate millions of electronic switches in order to perform logical and arithmetical operations on given input. Computers can only operate on a series of binary digits—bits—which are represented internally as on or off electrical states. From very simple operations—AND's, OR's, NOT's and other basic logical operations on 1's and 0's—extremely complex procedures can be constructed. The set of instructions written by humans that tells the computer which operations to apply to data is known as software.

Software was originally composed in the binary form that computers pro-

cess internally. Early on, however, it was realized that this form, while efficient for the computing system, is difficult and error-prone for human programmers. Thus, computer languages and programming environments were developed to ease the writing of software in a way that is faster and easier for humans to use. Over 200 languages have been developed, including such early ones as COBOL and FORTRAN, as well as today's more popular languages as C++, Perl, or Python.

Computer languages are often delineated by dividing them into a series of generations based on their historical evolution. The first generation of computer languages was machine language—writing directly to the underlying bits and switches that compose a computer system. Second generation language, known as assembly language, was developed in the 1950s and substituted a variety of symbols and operators in the place of the 0's and 1's of the raw code. Because each computer model had a different underlying chip structure, however, each system had its own flavor of assembly language, which prevented easy portability from one system to another. Third generation computer languages were developed with a focus on portability: not only might a programmer learn only one language for a variety of computer

platforms, but software written for one system could be "ported" to another system by recompiling the program. Compiling the code translates the high level, symbolic language into a machine language applicable to a particular hardware architecture. In this way, much of the underlying "messy" details of writing code was abstracted from the programmer, who could then focus on developing the higher level algorithms, interface, and procedure for the task at hand. In addition, the symbolic, highly lingual nature of the third generation languages allowed them to be processed by other computer software as well, enabling such programmer assistance as syntax checking or assisted generation of code structures.

Fourth generation languages take the concept of abstraction to an even higher level by defining libraries of software "objects" that can be reused to facilitate extremely rapid prototyping and deployment. The use of abstraction, libraries, and software "objects" has created a layering model of software development. Programmers do not need to worry as much about the lower layers of development—those that speak directly to hardware, or generate components of the GUI, for instance. Instead, they can focus on programming to the top-most layers, leaving the underlying details to the

pre-existing code base and development frameworks.

The abstraction of various components of software design into layers enables much more rapid development of new applications (standing on the shoulders of giants) by rapidly interconnecting previously tested software objects in new combinations. It also enables one to move code between various platforms with much greater fluidity. Because the underlying layers can be modified or replaced, so long as the "hooks" or interfaces to the upper layers remains the same, the same functionality is more easily achieved on a variety of disparate platforms. This layering of software development designed to insulate programmers from the intricacies of particular chip architectures and machine-level implementations has drastically increased code production and the resulting complexity of software systems.

2.1 The Critical Role Of Software

Together with the rapidly increasing cost/performance ratio available over the past 50 years, the rising functionality of software has enabled computing to expand into a wide variety of areas critical to the world economy. In fact, the combination of increasingly complex software and increasingly capable computing hardware has enabled a networked global information infrastruc-

ture that supports much of the world's political, economic, military, and social activities.

Today, computers, run by software, support and sustain critical infrastructure systems around the world. Software functions in a variety of mission-critical applications in which design flaws or operational defects could endanger lives, potentially affecting millions of people. For example, software forms the core of systems that monitor electric generation and power grids, monitor traffic flows at airport hubs and in public transportation, and facilitate computationally-intensive simulations that encourage scientific progress in a plethora of fields. In addition, software manages much of the flow of information—including financial data–in today's society. A breach of security or an error in code could mean that private, confidential information may make its way into the wrong hands, that information is replaced by erroneous or false data, or that funds are mishandled. These security holes can cause millions of dollars in damage.

In addition to managing critical infrastructure, the software industry itself plays a substantial role in the overall economy; software plays an increasingly important role in the National Information Infrastructure and is a major com-

ponent in the smooth functioning of the economy. Federal Reserve economists estimate that the production of Information Technology products—software and hardware—has contributed approximately $50 billion in productivity output annually since the mid-1990's, representing about 67% of the annual $70 billion gain in overall productivity demonstrated by US businesses during the same time period.[1] In addition, nearly half of American households use the Internet, and over 13 million Americans hold jobs in this sector—with a growth rate over six times faster than average.[2]

Given the critical role that software plays in today's world, software, regardless of its origin or process of development, must be evaluated in terms of its quality and merits. Poor software can have devastating repercussions, negatively affecting hundreds of millions of people around the globe.

2.2 Evaluating Excellent Software

> "Computer programming is an art form, like the creation of poetry or music."
> —*Donald E. Knuth*[3]

Evaluating software is problematic. What, exactly, is excellent software?

[1] Please see [5].
[2] Please see [6].
[3] Please see [7].

There are a variety of ways to evaluate "excellent" software, but there is no single generally recognized way to define or measure excellence. A variety of different methodologies have been developed to aid software management teams in ascertaining the end result of their programming tasks, and they encompass a variety of different perspectives on the entire software development process.

For example, one numerical approach entails counting the number of "defects" at each stage of testing, and dividing by some measure of the size of the project, thereby determining the defect density.[4] A particular problem with this approach, however, lies in determining exactly what a "defect" is. In addition, even if a particular piece of software contains no "defects" or deviations from its intended specifications, it may not be considered an "excellent" piece of software by its users; that is, it may be difficult to use, or it may not do what the customer thinks it should, even if it is designed according to the engineer's specifications.[5]

As this example demonstrates, as much as people would like to quantize

[4]Please see [8].

[5]A likely problem in this sort of situation is that the specifications were poorly delineated. An improperly designed set of initial specifications will almost always result in poor quality software. For an example discussion, see the DoD discussion of software quality available online at http://www.dacs.dtic.mil/databases/url/key.hts?keycode=3494.

their evaluation of software excellence for management and development purposes, it is difficult to assign a single value or range of numerical parameters to measure the "excellence" of a piece of software. In this regard, software, and many other areas of computer science, can be considered in Kuhn-ian terms as a "pre-paradigmatic" science.[6] Developing software is still more of an art than a strict engineering discipline, so it does not have universally acceptable means of quantitatively delineating "excellence." Yet, just as one can ascertain the excellence of a particular work of art, it is clear that some software is "excellent" and that some software is not.

To deal with this ambiguity, we shall consider five general criteria of excellence that are said to be required given the crucial nature of that software within today's world. Each of these criteria can be evaluated independently and may be more important based on the objective at hand, or on the purposes for which the software will be used. In addition, as in all engineering problems, there are a variety of trade-off's involved. Overall, however, these five perspectives of analysis contextualize a piece of software within a continuum of excellence.

[6]Please see [9].

2.2.1 Technically Competitive and Innovative

The rapidly developing field of information technology requires excellent software to utilize the best and most appropriate innovative techniques in order to be competitive. The technology world—and software in particular—has been in a state of flux and change since its inception in the 1940s.[7] Every generation of software brings with it changes in computing languages, design paradigms, reference models, and system-level theories. While many of the underlying algorithms and mathematical theorems remain constant, the structures and methodologies of programming evolve over time as new hardware becomes available, as network protocols and implementations develop, and as new theories make their way out of research and into practice.

In order to succeed in this environment, software must be technically competitive and innovative. Software, computing, and networking technology are developing extremely quickly; any code which does not compare favorably with its contemporaries in the market in terms of innovation will quickly fall behind and be eclipsed. Software must also be current so as to interoperate with other contemporary systems and information feeds as well as to provide

[7] One notes the slide in this military presentation: http://www.cascom.army.mil/automation/GCSS-Army_Global_Combat_Support_System-Army/MISC/coc/Council_of_Colonels_1998November05/09_Managing_Change/sld021.h

a timely, current analysis of the data under consideration. Interoperability itself is a trade-off, however, as cutting-edge innovation will always be in tension with establishing interoperability with other software and systems.

2.2.2 Tech Focus: Quality Results in Code

Balanced with the need for innovation and cutting-edge design, however, is the need for stability and operating quality. Because of the complex nature of software, it is quite easy for errors to crop up in the development of any software system. In the oldest computer systems, software programs were implemented by physically wiring connections between the various components of a computer; early mythology claims that moths flying amidst the connections would cause short-circuits and thereby introduce errors into the software applications. These "bugs" still persist today in errors that can be introduced at all levels of software design and implementation.

Bugs are generally more difficult to identify than traditional manufacturing defects. While bugs may be as simple as a typo or an unintended instruction that is readily discovered based on run-time testing, they are often more subtle, manifesting themselves in poorly designed logical structures that do not exhibit the error unless specific conditions in the data are real-

ized. Because of these intricate error conditions, software must be carefully evaluated based on its ability to perform its intended functionality efficiently, without error, and with as little "down-time," or inaccessibility, as possible.

Because software functions in concert with its users, truly good software must also be resilient enough to reduce possibilities for user error in its operations. Software must anticipate potential user confusion and be vigilant in its own testing of results and boundary conditions. In addition, the need to reduce user error is closely related to an additional criteria of excellent software, user interface design. Many programs are identified as being "buggy," when, in fact, it is user error creating the problem. The program itself may be operating without defect, but if the design is confusing and leads the user to operate it incorrectly, the design itself is effectively contributing to the errors.[8]

Not only must software be bug-free, it must also be secure.[9] The need for security to be incorporated into the design is tightly related to the need for

[8]This example demonstrates the difficulty with which criteria for evaluating software can be clearly compartmentalized. Bugs—often considered to be purely a function of engineering and technical details—are closely related to interface design, a "separate" criteria. Thus, it can be seen that the criteria for evaluating software's excellence is actually a contextualization of overlapping perspectives that cannot be completely isolated.

[9]Among 186 companies able to quantify their loss for the 2001 Computer Crime and Security survey found that various cybercrimes amounted to $378 million in losses during 2000. Please see [10].

software to be bug-free, as many security lapses result from taking advantage of flaws in a software system. Thus, good software is designed with security in mind, and it is thoroughly tested for vulnerabilities and weaknesses, both from malicious attacks and from unintended misuse. In addition, because no system is completely bug-free, secure software solutions are implemented in an environment so that damage from security breaches will have a minimal impact.

2.2.3 Operator Focus: Quality Results for End-Users

In addition to the stability, efficiency, and security concerns that are vital to the tech-centric "under the hood" side of software development, software must also be designed and implemented to provide quality results for individuals using it to accomplish tasks in a given operating environments. Thus, software must be driven by actual user needs and provide an accessible, easy-to-use interface design for the task at hand. For example, usability is the single criteria that business computer users most often link with quality.[10] For end-users, usability is equal to, and in many cases surpasses, the criteria of reliability, security, and performance. Indeed, if software is per-

[10]Please see [11].

ceived to be non-functioning because of its complexity, it may as well not operate at all.

As software becomes more complex, simply determining the goals and needs in coordination with end users can be a daunting task. For simple, single-task pieces of software, ascertaining these goals and implementing them in code can be fairly straightforward. Successful software exhibits a well-defined focus of attention that takes user expectations and delivers a coherent set of tools that matches these needs.

To meet user needs, software must possess an intuitive, excellent user interface that acts as an extension of the user's abilities, functioning in a way that does not impede accomplishing the user's goals. A successful interface is designed to aid the user and functions logically, cleanly, and efficiently. In the general software markets, interface appearance and functionality is a prime comparison factor between rival software products. In addition, as the use of software permeates both our society's personal and business lives, a well-designed, functional interface to information management increases productivity and user satisfaction.

In the user-interface arena, many studies have been performed with

usability-testing labs, and a variety of principles have been established. These principles provide guidelines for evaluating "excellence" in user interfaces, based on the results of examining thousands of users' interactions with computing systems. While these guidelines do not provide specifics as to how a user interface must function, they do provide criteria for determining the excellence of a particular user interface. Thus, though much of a user-interface may be considered "art," the fields of ergonomic design and applied psychology can highlight which designs are more functional and easier to understand, providing one criteria for judging "excellent" software.[11]

2.2.4 Excellence in Fully Utilizing Network Effects

Software is a classic example of network externalities leading to positive network effects at work. Network externalities refer to the condition whereby the value of a product to an individual adopter is directly related to the number of people who use the product. Software is a direct application of

[11]While standards have been established, graphical user interfaces still tend to be difficult to use because of "consistent usage but irregular implementation of spatial metaphors and spatial structures." [12] The GUI today is a mixed metaphor, having developed greatly from the original research at Xerox PARC through the Macintosh to today's GUI's.[13] The GUI's original relationship to spatial representations and metaphorical consistency has been replaced by an uncritical drive toward ease of use. An "ideology of ease"[12] has been constructed in which ease of use is the end goal, not merely the means to another goal. Such an ideology has profound consequences in maintaining divisions among different "classes" of users and groups. For more information, please read [12].

this concept; the more people who use an operating system, for example, the more likely it is that other people will write software that runs on it. Likewise, a larger pool of users makes it more likely that one will be able to leverage one's technical knowledge about that system to other people's machines. Application software and utilities also facilitate positive network externalities, allowing for the easy sharing of files, information, and user expertise.

The presence of network externalities leads to positive network effects, and produce positive feedback loops. Positive feedback is critical in high-tech industries such as the software market because of the concept of a virtual network. In a network—whether real, with physical connections between nodes—or virtual, such as in the hardware/software network of the programming industry—the value of being a part of the network is directly related to the network's size. This feature of networks encourages feedback loops wherein the strong get stronger and the weak, increasingly, get weaker or get eliminated entirely.

Successful software often utilizes the advantages of these network effects. To encourage wide-spread usage and extensibility, a piece of software should

have clear and easily accessible application programming interfaces, or API's. These API's define the "hooks" necessary to access the internal modules and data structures of a piece of software. If these hooks are easily accessible to programmers and clearly and publicly defined, the software can be plugged into a variety of other situations, and more developers are encouraged to write software applications that fit into this arena. Using this open API strategy, the software is able to take advantage of network effects; the more potential network "connections" that are available to the piece of software, through ties and links to other products and services, the more power is available to the user and the more benefit one may obtain by using the code.

2.2.5 Long-Term Economic Viability

Finally, a successful software systems project must be able to function within the current market framework and maintain itself economically over the long-term. In today's case, that market framework is one of global capitalism. Thus, software must be able to be self-sustaining economically and be able to finance its own current and future development. Developers must have motivation and capital to initiate and continue development of their software product.

Software exhibits extremely high costs of development, but low marginal costs. That is, the research, management, and programmer labor costs that go into developing a particular piece or class of software is generally quite high. Once the software is complete, however, it exists in pure digital form, and it thus costs virtually nothing to reproduce and distribute. This pricing and development structure makes it easy to version price software depending on the needs of, and value to, particular subclasses of consumers. Low marginal costs also mean that economies of scale don't tend to disappear when the market grows too large; the marginal cost for each new unit remains low instead of exhibiting the qualities of a step-function found in traditional non-networked industries where additional demand, after a certain point, often incurs additional production, maintenance, and organization costs.

Historically, software has been financed through a variety of approaches, which allowed for covering its initial high costs. Many early software projects were financed as academic or government-sponsored research. Once IBM became successful in promulgating computer development, it would often develop software as part of an overall systems package, the price of which would also include hardware and onsite support. In 1970, due to an antitrust

court decision, IBM began charging a separate fee for its software and thereby opened up the market for independent software developers to write their own software for computing systems. Thus, in this case, licensing of the software itself provided the revenue for future development. Besides licensing of proprietary code, other revenue models have included income from support or consulting or subsidized development with revenue incurred from associated hardware sales.

Regardless of the means, however, software must both generate sufficient capital investment to satisfy its research and development needs as well as provide motivation for its programmers. In many cases, the motivation is in intrinsic interest in the "art" of programming software, and programmers produce code as a hobby, with their livelihoods being paid by "day jobs." In this case, the "hobby" nature of that development track satisfies the needs of that particular software project in the economy. In any case, there still remains for any excellent, successful software system a need for financing the individuals who thereby participate to ensure its continued current and future development.

2.3 Conclusion

Programming excellent software is hard. In fact, over a quarter of all software projects are canceled before completion. The typical project is delivered over a year after the initial deadline, and most project arrive a full 100% over budget.[12] Not withstanding the challenges entailed, excellent software is increasingly critical in our information-centric modern world. Because of the extreme importance of software in supporting the global information infrastructure that undergirds the entirety of our political, economic, and social life, excellent software is a must.

Ultimately, the success and excellence of large software systems development projects depends in large part on the ecosystem in which these projects grow and mature. Software and programming, while exhibiting aspects of an engineering science, resembles more of an art to be cultivated than a process to be algorithmically delineated. Much as the environment in which a plant is cultivated can determine its pattern of future growth, so too the conditions under which software develops largely determine the parameters of the final product.

[12]Please see [14].

Despite the varying conditions under which software is developed, excellent software shares a set of common qualities. First, software must be innovative, competitive, and up-to-date. Second, the software must please the "geeks"[13] by producing quality code results that encourage bug-fixing, efficient algorithms and coding practices, and a keen focus on security. In addition, the software must please end-users by producing quality results that not only are driven by user needs and goals, but are also accessible through a well-designed and implemented interface. Excellent software internals are useless without an interface that provides maximum usability and efficiency for the target audience. Fourth, an "ideal" development environment must take full advantage of network effects by providing accessible, clear API's and hooks for developers, as well as by integrating and exhibiting interoperability with networks of all types. Finally, the software must provide a successful means of financing its own continuation and survive in the global marketplace. These criteria are the key to any successful software systems project.

[13] "Geeks" here is a technical term, which has been linguistically co-opted by the computing community from its original slang, insulting meaning of an "odd or ridiculous person". For more information about the transformation of the social group known as "geeks", please see [15].

3 The Incumbent: Software Excellence in a Closed Development Environment

Now that we have established an overall framework through which a software development environment can be evaluated for its excellence, we will consider the incumbent method of computer software development: the software development ecosystem that has developed around proprietary software. Herein, we refer to proprietary software as that software which is developed by a company for sale and/or licensing to other parties, and whose source code and future development is not available for full inspection, modification, and future distribution to the customer. Such software ranges from simple, one-task "shareware" programs, which authors may release through the internet, with online transaction fulfillment, to full operating systems and office productivity applications containing millions of lines of code, such as Microsoft's Windows and Office product lines. After examining how the history of software development has led to the preeminence of the proprietary development model, we shall examine this proprietary development environment with respect to our criteria for fostering software excellence.

3.1 History and Development

Early computing systems tightly coupled client consulting, hardware sales, tailored programming, and support and training. The very first computers had to be physically rewired for each new application or revision to the instruction set that they processed. Even after "software" became soft—that is, abstracted from the physical hardware wiring—software remained tightly coupled with a particular computer because each computer system was expensive and highly specialized. Gradually, the concept of a hardware architectural platform developed, wherein software could be written to an abstract specification, allowing it to run on other machines sharing a similar architecture. In spite of this development, software remained tightly coupled—it was written by programming consultants for a particular corporation or research institution, and the software itself was not separated from the total computing support package.

The primary reason for integrating support and custom-written software was cost: computing equipment was simply too expensive, and too specialized, to easily allow the proliferation of pre-packaged software solutions. Studies during the 1950's and 1960's demonstrated that users simply would

not use off-the-shelf packages for common business functions; the requirements were considered "too specialized." While the packages didn't change, these same companies embraced prepackaged solution less than two decades later. The difference lay in the hardware/software cost ratio: while the purchaser of a $2-million machine in 1960 could afford a $250,000 customized business application that integrated seamlessly into his current computerless environment, purchasers of inexpensive desktop devices could not afford customized applications, and instead fit the off the shelf packages into their workflow.[1]

The advent of the personal computer changed the way companies obtained revenue from software. As personal computers proliferated, computer enthusiasts began producing and sharing their own software for their own machines. Since the primary audience was fellow hackers, code and software were shared freely so that each would improve their own coding skills, and high quality applications and utilities would be available to all. At the same time, the wide availability of multiple machines running similar architectures, delivered to a mass audience, enabled companies of programmers to produce

[1]Please see [16], p. 198.

software that was general enough in purpose to be sold and distributed to a wide audience. Software, which was once tightly coupled with a complete hardware, design, and support package, could now easily be separated and sold to individual computer users, provided that distribution and licensing rights could be secured.

Because it could now be separated from the total revenue stream, software written for computer systems attained property value by itself—independent of the complete package that had heretofore delivered value. Software, the abstract notion of computing instructions encoded in mathematical code, became a piece of property capable of being sold—or stolen. The software industry views this software code as traditional property, covered by traditional intellectual property protection under which the producer retains complete control over distribution and usage.

The notion of software as a traditional property, "owned" by its producers, has resulted in a proprietary, closed model for software development and distribution, with a revenue model based on licensing of binary, compiled applications. While custom software is still common for large-scale enterprise applications, even these tend to use pre-packaged, commercially available tool

packages for development and execution. In addition, the code at the heart of these projects is also closed and proprietary, generally available only to the company for which it was designed and to the programming consultants who will implement it.

The 1975 "stealing" of Bill Gates's and Paul Allen's original BASIC software tapes during a Silicon Valley demonstration meeting illustrates the shift of software as a smaller portion of a larger revenue pie to becoming property valued in itself.[2] At the time, software was not usually considered an item with property value by itself. Bill Gates, however, had written the first BASIC for the Altair and had intended it to be sold by MITS (the makers of the Altair). In keeping with the open sharing of code common at the time, this BASIC software was quickly copied and distributed to a band of hobbyists in the Homebrew Computer Club, who felt a community obligation to spread the software as far and wide as possible. Gates's scathing letter denouncing the activity as "piracy" fired an early shot in the ensuing war over property rights in the new domain of the digital.[3]

[2]Please see [17].

[3]As Gates lambasted the hobbyists, "Why is this? As the majority of hobbyists must be aware, most of you steal your software. Hardware must be paid for, but software is something to share. Who cares if the people who worked on it get paid? ... Who can afford to do professional work for nothing? What hobbyist can put 3 man-years into programming, finding all the bugs, documenting his product, and distributing for free?"

The members of the Homebrew Club began to realize that as the total market size grew, software attained sufficient mass that it could be sold by itself. The actual "property" of software was valued separately from the support or consulting that had heretofore accompanied it; software was becoming an independent business. The production and control of the code at its core could give great wealth and power to those able to seize it. Like any new settlers encountering a vast new frontier that suddenly demonstrates the potential for great wealth, the participants in the burgeoning software industry became embroiled in a struggle over the structure and governance of property rights.[4]

3.2 Technically Competitive and Innovative

To get a foothold in a particular market niche, proprietary software must be competitive and innovative. To encourage customers to purchase a particular software package, the system must either enable the user to perform tasks that were previously impossible or too difficult, or offer an improved solution, either in terms of speed or user efficiency. In addition, sales of up-

Gates's letter did not produce many additional purchases of his Altair BASIC, but it did signify one of the early moves from a model similar to "Open Source" to the currently dominant model of proprietary licensing for software. For more on this critical moment in the history of the computer revolution, please see [18].

[4]Please see [19].

grades to current customers form a large part of a company's revenue stream; it is therefore in the best interest of the company to continually improve its product by offering new features to entice current purchasers. Finally, companies pursue cutting edge technology because radical innovations within a brand new niche—so called "category-killers"—lead to a dramatic first mover advantage that cements a firm's position within a specific market segment. Thus, market forces provide an incentive for software produced in a proprietary environment to be competitive and innovative.

Proprietary control over all aspects of a project often leads to a product whose individual components function together much more smoothly and thereby offers a more compelling user experience. The tightly-controlled nature of proprietary software development encourages direct, coordinated management of an entire project, which under a best-case scenario leads to a greater unity both of design and execution. This finely-honed management coordination focuses the efforts of a variety of programmers, thereby increasing effort and attention to less successful portions of the code, removing fluff, and encouraging developers to work on less "fun" or interesting, but still necessary, coding sections of the overall project. In addition, the

management-imposed integration allows for a degree of synergy: very often, taken together the disparate parts of a software project, when combined, enable innovations beyond their separate applications. The central-control aspect of proprietary software makes synergistic software innovation easier to obtain.[5]

Licensing of software provides a steady stream of revenue to fund future research and development. Microsoft, for instance, enjoys one of the highest margin rates of any corporation because of the low cost of distributing licensed software. On 2000 revenue of $22.9 billion, Microsoft spent $3.775 billion on Research and Development.[6] Such large revenue flows encourage a wide variety of research activities that can lead to new software innovations and break-through technologies in the future.

3.3 Tech Focus: Quality Results in Code

Producing efficient, bug-free, secure software has long been the "holy grail" for software engineers. There are two basic metaphors used to describe the development environment of quality software, both of which are

[5]The experience of Open Source coders is that each of these benefits that seemingly require tight management and control can also be obtained in a decentralized, bottom-up environment. Please see [20], pp. 70–75.

[6]Please see [21].

used by the proprietary software development community: "building" and "growing" software. The "building" of software views systems programming as a process by which one completely specifies a satisfactory system in advance and then proceeds to implement it. The metaphor of building leads to the use of other elements—namely specifications, assembly of components, and scaffolding which translate into activities used by software engineers.[7] In most real-world systems, however, it is nearly impossible for clients to completely specify the precise requirements of a complete product without ever having built or tried some version of the product being described. "Building" software has become too cumbersome in developing of software systems that are increasingly complex.

The alternative is to "grow" software. Software systems can be grown by incremental development—that is, the system should first be made to run with a skeleton of code, containing empty routines for anticipated functionality. Then it is gradually expanded and fleshed out" as subprograms are developed into operating units. Such an approach requires top-down design; the software literally grows from the top down, expanding to fill out its skele-

[7]Please see [16] p. 200.

ton. The model of growing software also lends itself to early and frequent prototypes, allowing for the fine-tuning of features and implementations, easy backtracking to previous states, and for the addition of more complex data or condition sets that grow organically out of conditions already established and coded.[8]

Building or growing systems software requires attention to detail and an overall, organized systems plan. Both components arise easily out of a tightly-controlled proprietary development environment. Given its top-centered management focus, a proprietary development environment is ideally suited to encourage a comprehensive structure for a programming project.

The single software group that most epitomizes the tight control associated with this mode of building excellent software is the shuttle programming group at NASA—an example of the potential for excellence within a closed, proprietary software group. The work of 260 programmers located across the street from the Johnson Space Center, this code is responsible for the launch and safe return of the $4 billion shuttle, the lives of six or more astronauts,

[8]Please see [16] p. 201.

and the expectations of a watching nation. This software is as near perfect as is possible in the software industry—declared by some to be practically "bug-free." According to the shuttle group's internal statistics, the last three version of the program had just one error each for over 420,000 lines of code. The combined last eleven versions had a total of only 17 errors. As a benchmark, comparable commercial programs would have an average of over 5,000 errors.[9]

To accomplish such "perfect" software, the on-board shuttle group has established four propositions that serve as guidelines for building software with quality. These propositions provide excellent criteria by which other groups organized around a proprietary development environment can pursue technical quality.

First, the product is only as good as the plan for the product. The specifications and functionality of the code are agreed to before any code is written; in addition, no line of code is changed without specifications being written that carefully outline the change and its reasoning. The group does the code right the first time, and don't change the software without changing

[9]For an overview of the quality computing performed by this mission-critical group, as well as the methodology of their programming practices, please see [22].

the "blueprint."

The second proposition is that the best teamwork is a healthy rivalry. In this "building" design of software, programmers are divided into a group of coders and verifiers. Each group reports to separate management and strives to outdo the other; each is in competition to find errors or holes in the code. The very presence of the verification group makes the coders more careful. The success of this friendly tension is evident: the shuttle programming group, for instance, now finds 85% of its errors before the formal testing period begins, and over 99.9% of bugs before the program is delivered to NASA.

The third proposition is that the database is the software base. The project itself sits atop two enormous databases: one contains the history of the code showing every line, when and why it has been modified, and what specifications in the original blueprint discuss the modification. Thus, the complete code "genealogy" is available to everyone on the project. The second database is an error database that contains a record of every error that has been discovered, including all details about its discovery, how it made it past error-checking filters, and how it was eventually resolved. The

database incorporates so much data, in fact, that models have been written to forecast how many errors should be produced in new versions; if the number of discovered errors is less than the projected count, code reviewing processes are redoubled until the number of errors found matches the prediction.

The final proposition for building quality software is that any mistakes are a result of an error in the process; the objective is to not just fix the mistakes, but also to fix whatever allowed the mistake to occur.[10] Thus, errors are not blamed on individual people, but on errors in the inspection process, or to the evaluation and testing process. Thus, the process not only finds errors in the end software product, but it also finds errors in the process itself. A proprietary development environment handles the structural requirements of such a process admirably; a hierarchical command and control structure, leading from the company executives through the managers to the individual programmers, can encourage attention to design and quality, and build the process by which excellent software is constructed.

In most large development projects operating under a proprietary devel-

[10]This methodology is similar to the Total Quality Control system used in Japan to implement "continue improvement" in which quality is developed not only within the product, but within the entire system that develops, produces, and sustains the product during its lifecycle. For an overview of TQC, please see [23] and its associated bibliography.

opment paradigm, market forces and the reality of a license-driven profit system can, and usually do, undermine this successful process. Speed to market—and the associated profits and competitive advantages that come from an earlier release—often drive concerns for total code quality to the sidelines. While the process is not extremely expensive—carefully planning the software in advance, writing no code until the design is complete, making no changes without support blueprints, and keeping an accurate record of the code—it does take time, and a dedication to overall quality in lieu of speed to market. Indeed, "The process isn't even rocket science. It's standard practice in almost every engineering discipline except software engineering."[11] Yet, making the process a formal part of software engineering within the proprietary, closed development environment often places other, profit-driven needs in tension with the drive for quality.[12]

3.4 Operator Focus: Quality Results for End-Users

The single biggest factor in obtaining quality for end-user interaction with a piece of software is to perform regular and repeated user testing during the

[11]Please see [22].

[12]For more information about how the incentive structure of US proprietary software companies drive them to produce software of poor technical quality, please see [24] as well as [25].

design and implementation phases. Iterative design, a repeating cycle of design and testing, is a validated methodology that consistently produces high quality results that—most importantly—are in synchronization with user needs and expectations.

User testing allows companies to fix problems before the product is delivered to the public, enabling a better perception of the product, as well as decreased costs for the software vendor. Iterative usability testing allows the programming team to concentrate on real, demonstrated problems, instead of "imaginary" problems that they may anticipate, but which are not causes for concern; developers often have no way to tell the difference between potential, real usability problems and those that simply are not issues. Usability testing provides them this tool. Finally, usability testing removes the guesswork from user interaction. As Bruce Tognazzini—noted user interface expert and contributor to the original Macintosh user interface design—commented:

> "We've all been to those project team meetings where perhaps ten $100/hr engineers, designers, and marketing people sit around and debate how users are likely to respond. That's $1000 an hour for uninformed opinion. One usability professional, applying the scientific method, can have a real answer in two hours for a tiny fraction of that amount. Not only that, it will be the right answer."[13]

[13]Please see [26].

Hyperbole aside, usability testing does finely hone the programming efforts of a development project to those issues with which the end-users have the most difficulty. [14]

Proprietary software development companies are well-suited to implement usability testing laboratories for their developing software products. Not only does a finely-crafted end-user interface improve their product, thereby making it more favorable to consumers and businesses in a competitive market environment, usability testing also saves money by focusing coding time and reducing time to market. Thus companies whose revenue streams depend on selling proprietary binaries and whose products exist in a marketplace with multiple, competing vendors have a financial incentive to engage in usability testing and maximize the user experience and interface of their software product. In addition, proprietary companies also possess the funds necessary for usability testing in a laboratory because of the revenue obtained from licensing.

[14]For an overview of software usability testing, please see [27].

3.5 Excellence in Fully Utilizing Network Effects

The tremendous propensity of a positive feedback loop in network industries tends towards a "winner-take-all" market.[15] In a networked system, there exist positive externalities to having a large user base connected to the network. That is, the value to a particular node of the network increases in direct proportion to the number of other nodes on the network. Thus, a particular network product can benefit from a positive feedback loop if it obtains a critical mass of users; the benefits of even more users joining the system increases in a rapidly growing fashion. Large players therefore become even larger, and small players in the market suffer from a negative feedback loop in that they decline in numbers as fewer people use their product. Once a positive feedback loop is in full swing, the market has "tipped" towards that player and normal network forces enable the victorious player to gain an overwhelming market share.

The software industry is extremely "tippy." Once a particular piece of software gains a lead, either through a first-mover advantage play or through successful marketing and deployment during the very early stages of a prod-

[15]For an overview of the economics of network industries, including case studies and strategies relating to network effects, "tippy" markets, positive feedback loops, and first-mover advantage, please see [28].

uct category's life cycle, it tends to vanquish all competition. This "tippiness" leads to dominance based on "demand-side economies of scale." In these situations, customers value the leading software system because it is most widely used. While opponents may be available, and even in some cases technically superior in terms of quality, they simply don't have a mass of users to outweigh the tremendous network externalities embodied in the dominant system. As Shapiro and Varian have commented, "if everybody else uses Microsoft Word, that's even more reason for you to use it too."[16] Users of software, just like users examining products in other network industries, want to be on the side of the "winner"—the product that will last and control the industry.

Closed development ecosystems are well suited to retaining control of existing positive feedback loops. Once a particular software niche has tipped—particularly if the software niche is a complex system that involves a variety of components operating as a platform upon which other software can be based—a software producer can maintain dominance merely through incremental improvements in the existing software and by fending off encroach-

[16]Please see [28], 180.

ment through the use of proprietary "hooks" or extensions that function only within its own system. Thus, a goal of many closed software producers is to obtain market dominance through the creation of a proprietary platform; such a platform, if successfully implemented, ensures a steady stream of revenue from upgrades and add-ons, because the particular market niche is "locked into" this vendor's software product.

To actually force the "tipping" of a particular software category, a software product must attain critical mass in the market place, a difficult task especially for proprietary software. One barrier to entry in the proprietary marketplace is that a critical mass must be generated by distributing products to users, and having them use them long-term. Because revenue for most closed software is obtained through product sales and licensing, only the largest, most well-funded companies can afford to give away a product in its initial days as a means of gaining the market penetration and ubiquity necessary to tip the market in their favor. Thus, the new product must be sufficiently better than its competition, be well-marketed and sold, and have significant bundling deals with OEM's and vendors even to be considered by corporate IS departments or the general consuming public. Such extreme

barriers to entry often prevent smaller pieces of software produced within a closed development ecosystem, or software produced by small companies without an existing base of capital, to capture enough of the market so that network effects can take hold.

3.6 Long-Term Economic Viability

To succeed in the long term, any successful production of intellectual capital must be able to sustain itself financially within the current market framework of capitalism. After all, the producers of intellectual property must be fed, clothed, and housed just as any other workers. While many useful endeavors may be subsidized by other areas of the economy (the National Endowment of the Arts' subsidy—through grants of taxpayer's money—of "starving artists," for example) the software industry as a large and important component of the overall post-industrial economy is generally viewed with an eye towards self-sufficiency.

The means for obtaining revenue from the production of software within a proprietary, or closed, software development model, is to "sell the bits." Through copyright law and licensing practices, companies are able to retain tight control over the distribution of the "bits" of their software, thereby pre-

venting unauthorized use and copying and reaping profit from each copy of the product in computer systems around the world. Piracy—or the unauthorized distribution of licensed, proprietary software—cuts into the revenue of companies engaged in this business model, and is thus institutionally discouraged through the creation of industry-wide consortiums such as the Software and Information Industry Association.[17] Making money from software—including making money by selling proprietary, binary-only software—is difficult. In fact, most software ventures fail.[18]

For those companies that succeed, selling software is extremely lucrative. Once a company has a well-received piece of software, margins on the sale of that software are extremely high. Because of its digital nature, costs for duplicating and distributing software is low, leading to extremely high profit margins for successful software. As the world becomes more networked, costs associated with the distribution of software continue to decline. Duplication of software on CD-ROM's is cheaper than earlier distribution on floppy disks, and the move towards network distribution and purely electronic licensing models further decreases a company's risks associated with manufacturing

[17] http://www.siia.net
[18] Please see [29].

and storing physical products. Indeed, software is one of the "purest" of commercial endeavors, obtaining its profit almost entirely from the "intangibles" of abstract bits and algorithmic instructions.

Microsoft is often considered to be the most successful company in developing a method for controlling its intellectual property and obtaining revenue from producing software. Indeed, from three employees and $16,000 in revenue in 1975, Microsoft grew until in 25 years they became the world's largest software company, with nearly 40,000 employees and $23 billion in revenue.[19] Thus, for those companies that succeed, the production of software in a proprietary, closed environment has yielded tremendous economic self-sufficiency.

Long term, however, a revenue model based on "selling the bits" does not mesh well with the underlying dynamics at work in the production and use of software. To examine the market dynamics at play in selling software, we must look at the economic value of software. The value of software leads to a system of incentives whereby the market produces monopolies in various niches that lack a focus on quality.

[19]Please see [21].

Software has two distinct types of economic value: use value and sale value. Use value is the value of the software as a tool for performing a particular function, such as an inventory control system, or a word processor's value in improving writing productivity. Sale value is the value of the product as a final, sellable good—the value obtained through "shrink-wrapped" or licensed pieces of software sold to the customer. Traditional software developers pursue a factory model in using licensing ("sales") fees to provide a revenue stream wherein the sale value of software is proportional to its development cost and its use value.[20]

In fact, most software is not written for sale, but is written for internal use. Most programmer-hours are spent (and paid for) by the writing or maintenance of strictly internally code that has no sale value at all.[21] Such software, while offering tremendous use value, would not garner revenue if it were sold on the open market; its applicability is too specific and its customization too great for it to be generally usable.

Just as most programmer's salaries do not originate from the sale value of software, neither is the sale value of software proportional to its development

[20]Please see [30], p. 142.
[21]Please see [30], p. 143.

cost. Rather it is related to the expected future value of vendor service. Thus, the sale price of a piece of software rapidly drops to zero if the company that produced it either goes bankrupt or announces that they will no longer be supporting the product. Regardless of the development cost of producing the software, the sale price is completely dependent on the future use and support that the customer expects to receive.

The perpetuation of the factory model for software revenues sets up a system of incentives that leads "to a winner-take-all market dynamic in which even the winner's customers end up losing."[22] In a typical software lifecycle, 75% of the costs lie in maintenance and extensions.[23] By charging a high fixed cost and low or nonexistent continuing support fees, the incentive structure works against the production of excellent software for the consumer. Because the software developer makes revenue from selling the bits, most efforts go into the production of the software and its marketing and selling. Support and assistance to the customer will generally not be profitable and will thus not be given an equal amount of attention.

[22]Please see [30], p. 147. For a complete discussion of this perverse market dynamic, please see [30], pp. 145f. Much of the incentive model discussed herein originates from Raymond's original analysis.

[23]Please see [30], p. 145.

In the proprietary development environment, incentives work against actual use of the sold software. Use increases support costs, thereby cutting into a revenue stream founded entirely on sales. Thus, based on this model, software that is well marketed but not actually used produces the most revenue for a company. In addition, the factory model does not work in the long run unless the market is rapidly expanding; otherwise, continuing development costs overshadow sale revenue.

As the product matures and sales slow, many vendors must abandon their product in order to cut rising costs, either by openly discontinuing the product or by making support hard to get. In either case, customers are driven to competitors because the software's expected future value—contingent on service, is at stake. Monopoly is the only long-term solution. Even strong second-place competitors fall to the niche leader due to the incentive structure set up by a combination of a factory, sales-value revenue model and a closed, proprietary development environment. Even in the case of a monopoly, however, the customer ends up with software produced without an incentive for quality.

3.7 Conclusion

As one might expect after 25 years of software development for personal computers, much of it programmed in closed, code-controlling companies, a proprietary software development environment does have the potential to create excellent software. The proprietary development environment provides the incentive and financing structure to support the production of technically competitive and innovative code. In addition, the controlled, managed environment of software engineering can support high quality code development by "growing" software with a comprehensive, quality-focused software engineering process. However, market forces and the drive towards a rapid speed to market often undermine a goal of total code quality in much commercial software. Proprietary software environments do encourage user-testing and a finely crafted interface, for competitive reasons. In addition, while they may provide barriers to entry for small, less well-entrenched firms due to the problems of rapidly expanding a software base with licensing costs, proprietary development environments do encourage the widespread use of network effects. Finally, closed software ecosystems have demonstrated their financial viability; successful companies can produce revenue streams large enough for

self-sustained growth.

4 The Challenger: Software Excellence in an Open Development Environment

An alternative to the traditional closed development ecosystem is the Open Source "bazaar-style" development model.[1] Historically, a proprietary environment has been the dominant means by which software has been produced and distributed to consumers and corporations. As demonstrated in the previous chapter, this model does provide a framework to produce some excellent software. Today, however, there is another option: alongside the historical trend towards a proprietary development environment one finds an increasing amount of activity devoted to the Open Source movement.

The Open Source paradigm adopts a fundamentally different approach to the problem of intellectual capital at the heart of software production, and a radically altered structure for innovation and development. Indeed, the approach is so radical that the entrenched incumbents have taken to attacking it on purely ideological grounds: Microsoft Windows operating-system chief Jim Allchin commented in an interview on 15 February 2001, "Open Source is an intellectual-property destroyer. I can't imagine something

[1]The metaphor of Open Source development as a "bazaar" originates with [31].

that could be worse than this for the software business and the intellectual-property business." Concerned that the Open Source movement could stifle innovation in the computer industry, Allchin concluded, "I'm an American, I believe in the American Way. I worry if the government encourages Open Source, and I don't think we've done enough education of policymakers to understand the threat."[2]

Herein, we shall consider these claims. First, we examine the history and development of the Open Source method of software development. Second, we evaluate its development paradigm based on our previously established criteria of environments that foster the growth of software excellence.

4.1 History and Development

Because of programming's roots in the academic community, long a proponent of the free sharing of information, software has its roots deep in freedom of speech and exchange of ideas. Developing software was often seen as research akin to developing new mathematical theories or other types of scientific endeavors, so software's development costs were subsidized by other revenue streams or academic grants. Programs as mathematical al-

[2]Please see [32].

gorithms were hard to distinguish from other types of pure thought theories that weren't "owned" in the traditional sense by anyone. Thus they were difficult to classify under traditional property rights structures. The small costs of replication and the "sharing" attitudes adopted towards the high creative cost of development led to an attitude among early software developers that software should be freely shared among community members for the benefit of all.

The prevailing form of property that developed in the software industry, largely through the efforts of new companies such as Bill Gates's Microsoft, was based on traditional copyright and licensing structures that allowed software programs to be "owned" and their usage governed by "licenses" that gave the licensees certain rights regarding the software's usage. This model succeeded in capturing the lion's share both of the market and popular perception about the "natural" way of controlling software and digital information. Likewise, this model was widely accepted in other "content" arenas, such as the pre-recorded music and video industries.

The original "hacker"[3] mentality that encouraged the free sharing of code

[3]The term "hacker" has varied considerably in meaning over the years, resulting in a completely different public image today than when it was first introduced. For a discussion of the term and an overview of the hacker culture, please see [33].

and information survived, however, and eventually developed into the Open Source movement. The Open Source movement is an outgrowth of the GNU project, initiated by Richard Stallman at the MIT Artificial Intelligence lab in 1984. GNU, which stands for the recursive acronym "GNU is Not Unix," was started because Stallman and others felt that the source code behind a software product should be free, and no one should be required to pay for software. Believing that source code is essential to the growth of computer science, these programmers were concerned lest guarding intellectual property in a proprietary fashion limit growth and innovation in the field and concentrate power in the hands of a few.[4]

The key philosophic concept espoused by this group is the freedom of the software—a freedom rooted in free speech. This belief in non-proprietary software drove a wedge between the academic and business communities. In the words of Chris DiBona,

> "English handles the distinction here poorly, but it is the distinction between gratis and liberty, as in Free as in speech, not as in beer.' This radical message (the freedom part, not the beer part) led many software companies to reject free software outright. After all, they are in the business of making money, not adding to our body of

[4]For more information about the Free Software Foundation and its GNU project, please see [34].

knowledge."[5]

Thus, there existed a conflict between "pure" information—information created in the name of scientific progress or discovery, as much of the source code was purported to be—and information for-profit—the central core of profit-generation in the new Networked Economy. Largely because of this tension, the GNU movement—embodied primarily in Stallman's Free Software Foundation—never achieved momentum in the business community.

Despite its lack of resonance in the business community, the free software model remained an alternative to the prevailing proprietary model. In 1998, a group of leaders within the free software community gathered to find a way to extend the free software development model into the business world. Out of this discussion, Eric Raymond and others developed a campaign to win "mind share"—a campaign in which a series of guidelines were developed to qualify software as "Open Source."[6]

[5]Please see [35].

[6]The development of "Open Source" as an outgrowth of the Free Software movement created a tension within the community that still exists. Stallman and other members of the FSF camp do not recognize Open Source because it supports the coexistence of open and proprietary software, the pragmatic position adopted by proponents of the Open Source model. Instead, Free Software proponents work towards a world in which all software is freefree in the sense of freedom, not cost. For an exposition of this view, please see Stallmans comments in response to the Allchin statements regarding "Open Source" as a threat to innovation, [36].

Basically, these Open Source guidelines sought to encapsulate the free software ideals in a way that was palatable to the business community. These guidelines give programmers contributing to Open Source projects "the right to make copies of the program, and distribute those copies; the right to have access to the software's source code, a necessary preliminary before you can change it; and the right to make improvements to the program."[7] Thus, programs that are classified as Open Source are completely open in distribution and in code content, in contrast to those developed under the closed, proprietary system. Anyone—not just the original authors or company in which the software was developed—may examine the code of an Open Source piece of software, distribute this original code, and make modifications to the code.

By defining and marketing these Open Source guidelines, the Open Source community successfully countered the public perception that "free" software (largely represented by Stallman's Free Software Foundation) was an anathema to intellectual property rights, was related to communism, and was an affront to capitalist profit ideals. In marketing terms, Open Source sought to rebrand this development paradigm, replacing the negative stereotypes

[7]Please see [37].

fostered by the FSF with positive stereotypes that would be particularly appealing to managers and investors in positions to make key strategic decisions with respect to technology. At the same time as this rebranding effort was underway, Linux demonstrated success as a completely Open Source operating system for the same personal computers that were dominated by Microsoft's Windows. These two public events, a rebranding campaign combined with a high profile success case, increased the visibility of Open Source as a viable, alternative software development environment.

4.2 Technically Competitive and Innovative

Traditional software development has been performed in what Eric Raymond has termed a "cathedral" environment; it has been "built like cathedrals, carefully crafted by individual wizards or small bands of mages working in splendid isolation, with no beta to be released before its time."[8] Traditional development focuses on innovation as a necessary component to retain competitiveness within a market environment; the driving force for proprietary software innovation is competition. In addition, traditional proprietary development environments support large research and development budgets

[8]Please see [20], p. 29.

supported by their large profit margins and attendant revenue streams derived from their licensing model.

By contrast, Open Source development—a striking example being the development of the Linux operating system—most often resembles a babbling bazaar with a seething cauldron of different objectives, approaches, and styles. Open Source development differs from traditional development environments in that it doesn't have a centralized, top-down management structure concentrated on bringing products to market. Instead, software in an Open Source development environment arises from the bottom-up; individual programmers, bound together by network technology and common passions, produce software based on their individual needs and objectives. A vast collection of independent actors, acting as individual nodes, collectively generate a complex structure—a complete software system. While it is hard to believe that a chaotic system might produce software which is stable, coherent, and even innovative, bazaar-style software development does provide an environment that encourages technical competitiveness and innovation.

The "bazaar" style of Open Source development is often derided by proponents of proprietary development as being relegated to cloning and improving

ideas that are currently "state of the art" in computer science, but incapable of pushing the envelope and supporting truly innovative work. One of the most common accusations compares Open Source development to "chasing headlights." Because it lacks a clear, top-down management thrust, it is assumed to be incapable of innovation. As described in a leaked internal Microsoft memoranda written about Open Source,

> "The easiest way to get coordinated behavior from a large, semi-organized mob is to point them at a known target. Having the taillights provides concreteness to a fuzzy vision. In such situations, having a taillight to follow is a proxy for having strong central leadership.
>
> Of course, once this implicit organizing principle is no longer available (once a project has achieved "parity" with the state-of-the-art), the level of management necessary to push towards new frontiers becomes massive."[9]

Specific examples of innovation within the Open Source world aside[10], the fundamental error in the view espoused by advocates of proprietary software is the assumption that a particular environment—whether the proprietary cathedral development environment or the bazaar-style open environment—can force innovation to occur. Innovation and insight—breakthroughs—tend

[9] Please see [38].

[10] Innovative ideas built through the Open Source development environment include such notable examples as TCP/IP, http and HTML (the web), and Beowulf clustering for massive parallel computing using inexpensive individual nodes.

to arise from individuals. The development environment itself serves to nurture and respond to the creative leaps of individuals, developing these insights into mature, full-formed solutions. Thus, as Raymond concludes, "Cathedrals and bazaars and other social structures can catch that lightning and refine it, but they cannot make it on demand. Therefore, the root problem of innovation ... is how to grow lots of people who can have insights in the first place."[11] In this regard, the "growing" environment makes a difference on the final outcome.

The Open Source environment is highly conducive to nurturing innovation. In the Open Source world, individuals can submit their ideas to thousands of others; those who are interested and share the excitement of an idea can cooperate in honing and developing it to final fruition.[12] Such a system provides the breeding grounds for thousands of ideas to germinate. Out of this complex, chaotic playing field, excellent ideas will flourish because of the excitement generated by a network of interested minds.

As a community, the Open Source world values the contributions of in-

[11]Please see [20], p. 257.

[12]One can contrast this to a proprietary development environment. In a proprietary software development shop, the individual programmer with a new idea must sell his idea through various layers of the management political hierarchy in order to "sell" it to obtain resources in order even to begin to pursue the idea.

dividual programmers as an indicator of group status. In this sense, it functions as a gift economy, where individuals are valued not by how much they possess, but by how much they "gift" or give away.[13] Creativity and development processes thrive optimally in a gift economy, where the ideas of creative individuals can be expressed and valued on their own merit. As Davis Baird states, "Creative work needs a gift economy. It is in the nature of the creative impulse to give to–and to take from–the creative community. This is a consequence of the fact that creative people stand on shoulders. Gifts received prompt gifts given."[14]

There does exist, however, a tension between the gift and commodity economies insofar as an open-ended dialogue opens between the demands of the marketplace (the commodity economy) and the needs of the gift economy. The Open Source development model attempts to negotiate this windy path between the need for financial support from the marketplace and business interests and the "artistic" and scientific needs of the gift-based economy of creative enterprise.[15]

[13]Please see [20].

[14]Please see [39].

[15]For a discussion of how creativity thrives in a gift economy, please see [39] wherein he dissects the tension between gift and commodity economies and the influence on the creative process using a specific case of scientific instruments as a historical reference.

Ultimately, science itself, as developed over the last several centuries, is a result of an "Open Source-like" development paradigm.[16] The scientific model is based on the free exchange of information. Sharing information in the process of discovery leads to the cross-pollination of ideas and the development of new structures that would otherwise not have been possible. By sharing ideas broadly in a widely disparate peer-review process, the eyes of many scientists may catch what a small group would have missed. Likewise in the computer science community, the openness of source code leads to a higher degree of bug-finding and a more diverse approach to problems-solving than can be achieved through proprietary solutions.

4.3 Tech Focus: Quality Results in Code

A critical component of the Open Source development model is the mantra "release early and release often." In the Open- Source bazaar environment, quick-turnaround releases incorporated tremendous amounts of user feedback and responded quickly to bugs and problems. Customers—and users—were treated as co-developers, using the internet as a tremendous

[16]For an examination of Open Source development as a special type of academic research, and a critique of Raymond's view of the Open Source development process as "too simplistic", please see [40]. For Raymond's response to this "critique of vulgar Raymondism" please see [41].

platform for networked development. A cathedral-style development model considers this philosophy bad policy, as early versions tend to be buggy and feature-incomplete. "Common wisdom" holds that subjecting users to these early versions only serves to irritate them. Within the proprietary development environment, programmers desired users to see as few bugs as possible. Thus, releases were infrequent, interspersed with intensive debugging between releases.

The release early, release often philosophy is based on the maximization of man-hours thrown at development; as often quoted in Open Source development, this has been formulated as "given a large enough beta-tester and co-developer base, almost every problem will be characterized quickly and the fix obvious to someone."[17] Or, as is jokingly referred to as "Linus's Law" on the net, "given enough eyeballs, all bugs are shallow."[18]

In this formulation lies the critical difference between proprietary models of development and Open Source environments: in the cathedral style of development, bugs and design problems are tricky problems that are examined

[17]Please see [20], p. 41.

[18]Rapid turnaround and the application of variety as a driver of evolution in a complex system do not completely obviate the need for sound software engineering practices in an Open Source project. For an excellent overview of Software engineering maxims and their applicability in Open Source projects, please see [42].

for months by a small, dedicated team. This team develops confidence that all problems are accounted for, and that the code is optimized, and then releases the product to the public. In the Open Source environment, however, the assumption is that bugs are a shallow phenomena, at least when examined by a thousand eager programmers who are exposed to each release. Thus, releasing early and often exposes the code to the maximum amount of scrutiny for bugs, design flaws, and security holes, as well as enabling it to rapidly respond and adapt to user feedback.[19] In a nutshell, Open Source produces quality code because of a massive, iterative system of peer review.[20]

The wide-spread peer review process of Open Source "bazaar" style development results in software that has few bugs and has efficiency optimizations resulting from the examination of a multitude of experienced programmers. More users find more bugs. As more users utilize a piece of software, they add a variety of ways too approach and stress the program, uncovering more

[19]Please see [20], p. 42.

[20]This is a specific example of the more general sociological concept of the "Delphi effect." The Delphi effect states that the averaged opinion of a mass of observers (equally expert or equally ignorant) is more reliable as a predictor than that of a single randomly-chosen observer. In the case of Open Source development environments, participants are not even randomly chosen, but are self-selected as those who are interested enough to use the software and learn about its internal workings, providing a pool of people with much higher technical competencies than normal. The use of such a large pool of resources enable the complexity of software to be tamed through parallel debugging. [find good reference discussing this for further reading]

variation and increasing its usability. In the Open Source development environment, many of the users are also developers, so each one approaches a particular bug from his or her own analytical toolkit. Thus, while the existence of a multitude of testers may not reduce the complexity of a difficult bug from a particular developer's viewpoint, it does increase the probability that someone's "toolkit" will fit the bug such that the bug is "shallow to that person."[21]

Widespread peer review also contributes to increased security. Computer security expert Bruce Schneier notes that security is not obtained by concealing security defects. Instead security is found by demonstrating the system publicly, allowing flaws in secure design to be exposed and thereby eliminated.[22] The aphorism, common in security circles, is that "security through obscurity is no security at all."[23] That is, hiding the underlying algorithms will not, in the long term, prevent exploitation.[24] The Open Source peer

[21]Please see [20], p. 44.

[22]For reference on secure design and an overview of cryptography, please see [43].

[23]Please see http://www.tuxedo.org/ esr/jargon/html/entry/security-through-obscurity.html.

[24]In a networked environment, security is of absolute importance. However, many claim that a complete openness of protocols can lead to "spoofing" of client information. A sufficiently talented programmer could make another client that represents itself to a server as something that it is not, and use this access to cause mischief. On closer examination, however, the ability to spoof is based on a more fundamental flaw in the security model and could also be implemented by sniffing or other nefarious means if the source were closed. For a cogent deconstruction of this problem in the light of hacking online games

review process encourages widespread critique of all security techniques, enabling flaws to be examined within the context of the global security community.[25]

4.4 Operator Focus: Quality Results for End-Users

Open Source software is intensely user-focused, but the first-tier of users are often co-developers, or users with a bias towards technical interests. The usability patterns of this group do not necessarily match that of the more general computer-consuming population. While software developers have historically excelled at writing interfaces for other technically-focused users, they have tended to produce poor interfaces for "common" use, simply because this group was not included in their target audience.

The standard procedures of Open Source development can also be applied to the realm of user interaction and interface design. By releasing early and often to a wide group of users—including "non technical" users, interface designs can be rapidly promulgated across the user population, and the good ideas winnowed out from the chaff. One of the key components of Open

to "cheat" as an example of the broader problem of security through obscurity, please see [44].

[25]While openness is a necessary condition for networked security, it is not sufficient. For a thoughtful examination of ways in which Open Source is not necessarily a panacea, please see [45].

Source software is that it is written to "scratch one's own itch." The "itch" of usability is large, and it provides an "interesting" problem to a large class of Open Source developers. Open Source software can also take advantage of industry-designed usability testing programs, utilizing the scientific results of these sampling and testing methodologies to add to the data of wide-scale peer review feedback.

The distributed nature of the Open Source "bazaar" development environment encourages a less centralized interface model. Traditional environments encourage large, monolithic applications that are defined by their user interface, directed by top-down project design and management decisions. Bazaar-style applications, however, often separate the engine that performs the actual "work" from the user interface. This modularity of design increases reliability by decreasing the size of individual components. In addition, the separation allows the user interface to evolve independently from the underlying engine that performs the logic of the application. The component approach also gives rise to multiple interfaces to the same core each of which may be customized to the needs of a particular client or distributed across a network in an enterprise or internet setting.

4.5 Excellence in Fully Utilizing Network Effects

Open Source development fundamentally hinges on an open code base—on the "commoditization" of software. That is, it is based on openly defined services and transparent, available protocols. This openness is good for the consumer—it provides competition and choice. Openness also allows for rapid deployment; later products can be built on earlier ones, enabling rapid construction of a towering infrastructure that may not have even been envisioned by the original authors.

The HTML/HTTP specifications provide an excellent example of how the open software development environment takes advantage of network effects. Because these two protocols were released to the world without restrictions, other development teams were quickly able to build competing implementations that remained interchangeable due to the standard nature of the base protocol. While certain extensions (the early Netscape/IE only pages in particular) remained proprietary, the benefits to consumers of interoperability outweighed the competitive edge gained by a particular proprietary implementation, and a plethora of products based around this open standard emerged.

The commoditization of software removes one of the primary means by which traditional companies retain control over markets generated by network effects—proprietary protocols. In order for companies to retain control over their products, they must introduce protocols and formats that only they control. This enables them to lock customers into their particular operating system, because moving to a competitor would involve high switching costs.

Such a strategy is evident in the proprietary development world's response to the Open Source movement. For example, in Microsoft's infamous internal "Halloween" memorandum outlining the Linux "threat", Microsoft claimed that "Linux can win as long as services/protocols are commodities." Thus, to counter the loss of control (and hence revenue), Microsoft (and other proprietary companies) determined to "de- commoditize" networks and protocols by "folding extended functionality ...into today's commodity services."[26] By "embracing and extending" existing open protocols, proprietary companies can wrest control of these protocols away from the competitive, open playing field and into their own development circles.

[26]For the complete text of the original internal memoranda, as well as associated commentary, please see [38].

Open Source development tends to resist such a controlling strategy in favor of simple, common protocols that enable a layered approach to future programming development. Protocols are designed to cleanly and simply connect parts of a piece of software's functionality, isolating code into manageable units that may be maintained and advanced in parallel. API's—Application Program Interfaces[27] —are open, documented, and clearly-defined. There are no "secret" API's, because the code itself is open for inspection, and any piece of software can easily connect to other pieces.

The openness and easy availability of connections to other software enables Open Source-developed software to take full advantage of network effects. Robert Metcalfe, inventor of Ethernet, once observed that the value of a network increases as the square of the number of nodes connected.[28] This observation can also be generalized to encompass software design: the value of a piece of software is directly proportional to the square of the number

[27]API's are an abbreviation for Application Program Interfaces. They are a set of common functions (subroutines, modules of code), protocols (specifications for communication), and programming tools. API's are designed as "hooks" for programmers to use when writing applications so that they can leverage an existing code base, or library of pre-written software. By using a set of common API's, programmers not only save development time by not having to reinvent the wheel, but they also ensure that their applications share a consistent user interface. Please see http://webopedia.internet.com/TERM/A/API.html for more information.

[28]Commonly referred to as "Metcalfe's Law." For background, please see [46].

of its connections. Thus, the clear API's and modularity that are byproducts of an open development environment contribute to a greater number of interconnections than that afforded by closed software which, given its secretive, controlled nature, limits code connections. The result is a code base that grows in interconnections rapidly, taking advantage of network effects to maximize software value.

4.6 Long-Term Economic Viability

To be successful long-term, a development model must be able to sustain itself financially. From this perspective, Open Source seems almost like magic: high-quality software emerges from the hands of dedicated volunteers around the world without any apparent financial motivation. Closer examination, however, reveals that Open Source does have a stable economic foundation, albeit one that differs dramatically from the licensing revenue model upon which proprietary development environments are based.

The pricing structure of Open Source software more accurately mirrors the cost structure of software development by producing long-term revenue based on a continuing exchange of value between the company and the customer through the use of service contracts, subscriptions, and other value-

added interactions. Instead of weighting funds exchanged at the front end, a continuing exchange allows revenue to be distributed over the lifetime of the software. An incentive is in place for quality software through a product's lifetime; bugs and features during the mature period of software development can be added not because they are marketable as forced "upgrades" to the existing userbase, but because they are value-added quality additions that are generated as part of an ongoing service relationship.

In addition, because the code is available to multiple parties, the future service value of the software is not dependent on a single vendor. Indeed, because a company retains the complete source code, it could contract future development out to a variety of other parties, regardless of the code's "abandonment" status by its original group of developers. Opening the code results in multiple vendors' competing both for support and software quality.

The shift to Open Source only threatens the sale value of software; use value—which represents most of the value of software—is unaffected. Indeed, open sourcing software is beneficial for its use value because it allows for cost-sharing of code as well as spreading the risk of the development across a larger community. For common applications, code is often reusable across multiple

knowledge domains without affecting sensitive company-specific procedures or information. In these cases, it is a tremendous benefit to the economy as a whole if the relevant code sections are shared and perfected as a common good, rather than being reinvented, albeit poorly and incompletely, each time the particular problem is encountered.

Though direct sale-value models are affected by a shift towards an Open Source revenue model, there are a variety of methods for capturing the indirect sale value of software. One is the use of Open Source software as a loss-leader to gain first-mover advantage for a later piece of proprietary software that generates a direct sales revenue stream. Open Source clients could pave the way for proprietary, sellable server software. For example, Freely available SMB clients—a type of file sharing—can be used on a variety of platforms to encourage the sale of proprietary SMB servers with which they can operate.

Open Source can also be used to create a market niche not for proprietary software, but for services. In this case, the vendor supplies additional value to the collection of bits as a service to the customer. For example, Red Hat sells a collection of freely-distributed software that is coherently packaged,

marketed, and supported. While the same software could be downloaded—for free—off the internet, there is added value in having a shrinkwrapped copy. People get the convenience of having a pre-packaged collection with a unified installer, confidence in a known brand, and the paper documentation and support services available from the company. In addition, Red Hat may offer other fee-based services, such as an online software update program or remote storage and backup, that tie directly into their Open Source software.[29]

Other companies use their Open Source products as a window into their consulting services, selling the brains and expertise that enabled the original creation of the software in an advisorial position. Although the software that they produce is freely available, and may be modified by anyone, having access to the original designers and programmers is often a major selling point in providing consulting and information technology implementation services. The Open Source product demonstrates the quality of the programming team available at the company. Clients evaluating the particular software product would consider it more efficient to hire the experts in a consulting role than

[29]For a comprehensive examination of Red Hat's value-added model for producing revenue from Open Source software through branding and services, please see [29].

to build the experience in the particular software system in house. Thus, the Open Source nature of the product opens doors and provides free marketing for a company's software.[30]

Additional revenue available from an indirect sales method included accessories, including everything from mugs and t-shirts to emphasize a brand to professionally written and edited documentation and reference works. O'Reilly Associates, for instance, produces a series of reference and educational volumes on Open Source software. In order to build its reputation in a particular niche, as well as encourage future development, O'Reilly even hires prominent Open Source programmers, including Larry Wall, the originator of the Perl programming language, and Brian Behlendorf of the Apache web server project.

Finally, companies could Open Source the client software and produce revenue on content delivered through this distribution channel. Software is merely a vehicle by which content is delivered on a subscription basis.

[30] An example of this type of revenue creation is Cygnus Solutions, who generated revenue off of contracts related to the Open Source gcc compiler. Cygnus did not sell the message that their engineers were "better" than the companies they were contracting with, but rather that there was mutual benefit for a company's outsourcing porting, support, and maintenance work to Cygnus because of Cygnus's already established reputation and expertise in the development of the compiler. For a business examination of Cygnus, please see [47].

Because the software is Open Source, programmers can port the client to other platforms and automatically grow the user base, providing access to more people to purchase the content.

4.7 Conclusion

The Open Source "bazaar-style" development model is a formidable challenger to the traditional top-down, "cathedral-style" development process of proprietary software development environments. Based on a bottom-up development process wherein complex systems emerge from a chaotic, intensely networked playing field of autonomous actors, Open Source development proceeds on a counter-intuitive path to attain its goal of excellent software. Yet, Open Source satisfies our previously established criteria of a successful software development environment. It fully nurtures innovation, often encouraging it in a greater fashion than a closed development environment. In addition, its reliance on a massively parallel system of peer review produces code that is of a consistently high quality, both in terms of technical excellence and efficiency as well as from a security standpoint. While developing a quality user experience has heretofore been a challenge, Open Source has the potential to apply the same peer review and improvement process to the

challenge of interfaces. This development model also takes full advantage of the networked nature of the current computing infrastructure, leveraging open standards, accessible API's, and common protocols to rapidly increase the value of each of its software parts beyond their solitary levels. Finally, we have demonstrated that the revenue models of Open Source development, far from putting programmers out of business because companies are giving away software "for free," are actually better suited to the long term needs of software maintenance and use value to the individual customer than are the "selling the bits" revenue streams of traditional proprietary development companies.

5 The Desktop Operating Systems Face Off

> "I think the desktop is king. It's the harder market to enter, but it's the one that tends to encircle and overtake the business use."
> —*Linus Torvalds on the Desktop Environment market.*[1]

No market niche for software is currently more visible or hotly contested than that for desktop operating systems environments. The modern desktop is the experience that most users have with computers. Its usability, feature set, and stability determine the general conception of computing for many people. In addition, the desktop environment is the platform on which other applications are constructed and deployed, thereby providing a "network" that enables other software to thrive or fail.

The desktop space is also one of the most difficult software niches to enter in today's market. Currently, the Microsoft Windows operating system runs on more than 90% of the world's personal computers.[2] Due to the network nature of desktop software, such an overwhelming "tipping" of software makes it very difficult for new entrants to compete with and succeed against the market dominator.

[1] Please see [48].
[2] Please see [49].

The importance of this software niche and the high barriers to entry combine to make desktop operating systems an ideal case study for the Open Source development model. Open Source software is least likely for this case: the opposition to a new entrant's success—regardless of its development model—is high. The immense size and complexity of a typical desktop environment–often consisting of hundreds of related programs and modules–is an additional barrier to successfully generate high quality software. If the Open Source development environment can successfully produce excellent software in this arena, it is likely to be able to generate excellent software in any context.

Herein, we shall first examine the current market leader, Microsoft Windows, and then take a closer look at the two dominant competing desktop operating environments available for Open Source platforms, GNOME and KDE. As we consider each software system, we will examine its background and evolution from a historical standpoint, and then proceed to evaluate its excellence and potential based on our previously established criteria.

5.1 The Incumbent: Microsoft Windows

5.1.1 History and Development

The success of Microsoft's Operating System Windows points to the extreme advantages that a software company, operating in a positive feedback cycle, can employ if it possesses first-mover advantages. By establishing a licensing agreement with IBM for its first personal computers, Microsoft began shopping its PC-DOS (later MS-DOS) with all new IBM PCs. In addition, due to a fortuitous business arrangement in its dealings with IBM, Microsoft was also able to license MS-DOS to any computer manufacturer that desired an operating system. Because IBM had created its PC with openly-available components instead of developing proprietary parts in house, the industry rapidly exploded with PC "clones." Most of these clone manufacturers then went to Microsoft to obtain an Operating System. Because Microsoft was among the first movers, and positioned itself so as to maximize the distribution and proliferation of its software, it rapidly achieved market-share and mind-share dominance in the burgeoning personal computer industry.

By incorporating ideas originating in Xerox's PARC and later employed in Apple Computer's Lisa and Macintosh computers, Microsoft continued

in its dominance with the announcement in 1983, and later release in 1985, of its Windows graphical user interface.[3] While it took several versions for Windows to attain wide-spread acceptance and a sufficient set of features[4], Microsoft's Windows gradually took center stage as the primary operating system among personal computer users. This prominent position was due primarily to Microsoft's widespread third-party hardware and software support, its extensive software development environment, and the preloading of its software on vendor's hardware systems. By the end of the 1990s, Microsoft Windows operating systems ran on over 90% of the world's personal computers.

5.1.2 Technically Competitive and Innovative

The resources and cash flow available from the successful licensing of software enables Microsoft to invest heavily in research and development and thereby develop innovations that improve its existing product line.[5] Mi-

[3] A timeline of Microsoft Windows history is available online at http://www.worldowindows.com/wintime.html.

[4] For example, Windows did not support overlapping windows until Windows 2.0 shipped in 1987.

[5] In its recent anti-trust hearings with the US government, Microsoft has often trumpeted its desire to innovate without the encumbrances of government hindrance. For example, Microsoft has established a "grassroots" organization called the "Freedom to Innovate Network" to support its advances to computer science. With respect to the anti-trust case, the innovation is the tight coupling of the web browser with the underlying operating system.

crosoft Corporation runs one of the largest research and developing arms in the world, spending over $3.775 billion on Research and Development in 2000.[6] Microsoft Research employs over 500 researchers, and has labs in China, Cambridge, Redmond, Washington, and San Francisco. Established in 1991, the Microsoft Research group is designed to investigate new technologies and processes and work directly with product development groups in order to introduce innovations directly into the Microsoft product line. Recent developments from this team include the development of an IPv6 protocol stack that will be integrated in Windows XP—the next version of Windows projected to be released during 2001—as well as advancements in cryptography, streaming media, data mining, text-to-speech functionality, and wireless networking.[7]

To stay competitive, Microsoft must develop or purchase innovations within the computer field and exploit them in its Windows product line. By integrating other technologies into its ubiquitous Windows desktop environment, Microsoft can streamline existing solutions and provide greater software coherence in the form of seamless interoperability. Because Mi-

[6]Please see [21].

[7]For more information, please see http://www.microsoft.com/presspass/features/2001/mar01/03-08collaboration.asp.

crosoft maintains control over the individual components, it is easy to deploy new protocols and applications by leveraging the existing userbase of Windows products.

5.1.3 Tech Focus: Quality Results in Code

Microsoft's strategy with Windows has traditionally been to bring new versions with new features to market as quickly as possible and not to fix bugs because bug fixes are not a significant generator of revenue. People tend to purchase upgrades based on the availability of new features, not based on improved stability or a reduction in the number of bugs.[8] Because of this incentive structure, Microsoft programmers have tended to focus more on increased features and less on bug finding.[9]

Because Microsoft makes its revenue by selling new features to users and users tend to have invested heavily in the Microsoft platform, generating substantial lock-in and high switching costs, the security aspects of the software

[8]For comments on the Microsoft perspective, please see the 23 October 1995 interview with Bill Gates in the German weekly magazine FOCUS, available online at http://www.cantrip.org/nobugs.html.

[9]For a discussion from "the other side" please see the well-thought through thread available online at http://db.tidbits.com/tbtalk/tlkthrd.lasso?-search&-database=TBTalk&-layout=ThreadList&-response=tlkResults.html&-op=eq&ThreadID=1202&-sortfield=GMTDate&-sortOrder=ascending. In this thread concerning software quality issues, an internal Microsoft employee discusses Microsofts focus from an overall project management perspective.

in question tend not to be a primary focus, despite multiple virus breakouts over a period of multiple years.[10] Thus, security design decisions have also suffered because of the incentive structure inherent in the Microsoft production of code. Not surprisingly, Microsoft systems and applications have a reputation for implementing inadequate security measures in favor of ease of use at the beginning of the learning curve or time to market.

For example, many of the most recent spate of email virii only infect Microsoft's Outlook application because of the method by which it naively "trusts" foreign code. Thus design decision enables any application access to control and modify sensitive files, including the user's address book, allowing rogue scripts the ability to replicate and be distributed to a user's entire address book. Due to Microsoft's revenue-based incentive model, however, the company has a greater imperative to produce new features and revisions than to investigate "minor" security issues.[11]

[10]Please see [50].

[11]Note that these security issues are not necessarily minor. One of the largest rash of server break-ins, mainly at financial institutions, took advantage of a larger number of Microsoft vulnerabilities. While security problems can be a problem for all operating systems, the particular incentive model inherent in Microsoft's monopoly, proprietary development structure is particularly discouraging for a proactive security posture. For a balanced analysis of the financial institution server attacks, please see the overview available on artstechnica.com available online at http://www.arstechnica.com/wankerdesk/01q1/greathack-1.html.

Because Microsoft operates in a proprietary development environment, its coding process is not generally open to public inspection and it thus practices security through obscurity. Security through obscurity conceals dangerous security flaws and limits the number of people who may spot potentially compromising coding practices. This coding method increases the chance that a particular security flaw will go undetected, and thus uncorrected. Taking a page from the Open Source movement, Microsoft has recently opened up some of its source code to select vendors in an attempt to provide greater peer review. This openness, however, is still limited in scope and does not take advantage of the massively decentralized complexity evident in the Open Source model.[12]

5.1.4 Operator Focus: Quality Results for End-Users

In addition to the more standard usability testing, Microsoft employs a customer feedback loop to evaluate its software interfaces. The delay between the stimulus and the response, however, is on the scale of multiple months to years as Microsoft releases subsequent versions of its Windows interface. Over the course of multiple versions and a lengthy iteration cycle, Microsoft

[12]Please see [51].

Windows has become progressively easier to use.[13]

As the interface employed by millions of computer users around the world, Microsoft Windows has become a de facto "standard" for graphical user interfaces, despite its contradiction of basic design principles.[14] Part of the success of the standardization of the Windows interface results from Microsoft's attempt to document a set of "guidelines" for designing Microsoft Windows environment applications.[15] While not as rigidly enforced as Apple Computer's equivalent Human Interface Guidelines in guaranteeing consistency of look and feel for application development, these guidelines do provide a base from which to build software that operates in concert with the environment and with other applications.

5.1.5 Excellence in Fully Utilizing Network Effects

The rise of the Microsoft Windows operating system is one of the most powerful examples of network effects. Microsoft has succeeded in almost completely dominating the market for personal computer operating systems

[13]For an overview of the interface of Windows XP (Whistler) compared with interfaces of GNOME 1.2 and KDE 2.0, please see [52].

[14]A collection of accessible essays on the basics of interface design, and the problems encountered by software designers, can be found at "AskTog: Human Interface Evangelism and Practical Design" available online at http://www.asktog.com/menus/designMenu.html.

[15]These standards are available online at http://msdn.microsoft.com/library/books/winguide/welcome.htm.

by taking full advantage of its first mover status on the PC platform and creating an enticing platform for developers. As the number of users of its system increased, a classic positive feedback loop was generated and it became increasingly difficult for competing entrants to gain a foothold in the market.

Microsoft has also successfully leveraged dominance in one software segment to extend it to other segments. For example, Microsoft was able to take advantage of the tipped market it had with the MS-DOS operating system to drive wide-spread acceptance of its Windows operating environment. Once the Internet gained ground in the mid 1990's, close integration of the Internet Explorer browser with underlying components of the Windows environment allowed Microsoft to increase market share for its internet applications.[16]

5.1.6 Long-Term Economic Viability

By demonstrating profitability and continued product development over the course of its product lifetime, Microsoft Windows has been able to sustain itself long-term. Long-term upgrade revenue is virtually guaranteed to Microsoft because of its near-monopoly over the pertinent market niche and

[16]Please see [49].

its ability to successfully leverage the platform-based network effects inherent in the software industry.

At the same time, notwithstanding a guaranteed revenue stream from upgrades, long term problems remain with financing the continuing development of an operating system. Pricing based on licensing does not take into account the continuing costs of software maintenance. To maintain a steady stream of revenue, Microsoft must release major upgrades of its Windows environment every few years with enough differences and additional "features" to entice users and corporations to upgrade.[17] However, as products mature and the market becomes saturated with a particular product, it becomes increasingly difficult to finance development with up-front pricing.

Recognizing the disconnect between the revenue stream and actual development costs, Microsoft has begun work, with the introduction of Windows XP, on a "subscription" service as an alternative to its traditional licensing arrangements. This scheme requires customers to pay for the software for a predetermined period, after which they must renew their subscription in or-

[17]Large-scale marketing campaigns that accompany new releases, such as the massive blitz accompanying the introduction of Windows 95, also help to raise the publics awareness and encourage upgrades based on brand recognition rather than an actual need of new capabilities.

der for the software to continue to function. Upgrades and service patches are included in their standard subscription fee. This pricing arrangement, while untested in the long run, has the potential to smooth the incoming revenue stream for Microsoft, allowing it to more easily finance future development.

5.2 The Challenger: KDE and GNOME on Open Source Platforms

5.2.1 History and Development

The Open Source world found its OS in Linux. While Richard Stallman's GNU project—to build a complete base of UNIX utilities and shells around a Free Software code base—had been in existence since the early 1980s, the GNU OS kernel—the HURD—never got off the ground. It wasn't until Linus Torvald's 1991 release of the Linux kernel that the Free Software world had an Operating System of its very own.[18] Because of Linux's roots in Unix, and its development focus on hackers and software developers, Linux remained a complex system without a unifying graphical user interface accessible to average consumers.

Early development of the graphical X windows environment had produced

[18]For an excellent overview of the history of Open Source, please see [53].

a robust windowing system, but X Windows by itself lacked a "toolkit"—or a single set of software pieces used to construct "widgets"—items such as buttons and check boxes that provide the means by which a user interacts with an application. As the Windows GUI became more popular, first with the advent of Windows 3.1 and later with Windows 95, two competing GUI toolkits emerged in the Linux world. A small Norwegian company called Trolltech produced the Quasar Toolkit—or "Qt." Although a commercial program, this software was available free to any user producing free applications on an X Windows system. In addition to the Qt graphical toolkit, another group of student programmers began producing the toolkit Gtk, and later Gtk+. This toolkit—the GIMP Tool kit—was initially developed to support the development of these students' image manipulation software known as the GIMP. As the software became more mature, however, it grew beyond a toolkit library designed to support a single application, and it became a full-fledged general purpose GUI toolkit. This software was released under a free, Open Source license.

Using the Qt toolkit, Matthias Ettrich released in October 1996 a document containing his ideas for KDE (the K Desktop Environment), which

he intended to be a complete desktop operating environment for Linux that would provide "a GUI for end users."[19] Almost immediately there was opposition to this proposal—based primarily on the usage of the Qt libraries, which were based on "non-free" or proprietary code, and whose source, at least initially, was not even available. In dealing with this onslaught of protests, KDE proponents made the conscious decision to proceed forward, founding the KDE project on Qt for practical reasons: they considered the Qt libraries to be the best technical solution to meet their needed goals. Ettrich, ever the pragmatist, considered the "non-free" license of the Qt a moot point. As he noted, "There's basically two kinds of software...good software and bad software."[20]

For Free Software fans, including Stallman and his followers, the two types of software were free and proprietary. The technical issues surrounding a piece of code were secondary to the fundamental notion of that code's liberty. Because Qt was not released under the GPL—even though KDE itself was—rights to modify and redistribute the code to Qt were not permitted. Under these conditions, Free Software proponents envisioned a future in

[19]Please see [53], p. 258.

[20]Please see [53], p. 259. For an internal look at the licensing decision, please see http://www.kde.org/whatiskde/qt.html.

which KDE succeeded, and the open GNU/Linux OS was simply a platform to run proprietary applications built on Qt and KDE, a scenario that was highly undesirable.

Once the attempt to convince Trolltech to use the GNU GPL for the Qt libraries had failed, Free Software proponents undertook a project to create a new GUI system based entirely on Free Software. In so doing, they adopted the Gtk+ toolkit created by Peter Mattis for the GIMP as the widget library. This new system would be known as GNOME—the GNU Network Object Model Environment—and would encompass an entire operating system that not only included a functional, easy-to-use user interface, but also a broader component-based strategy, taking the modular approach adopted in the underlying components of the Linux kernel and subsystems and applying it to high level, user accessible applications.

Eventually, uncertainty within the software community over the free future of Qt led Trolltech to develop a new license for its libraries—the Q Public License. Drawn up in close coordination with Open Source leaders, Bruce Perens and Eric Raymond, the QPL was released in November 1998. It addressed the licensing issues of most members of the free software community.

Finally, Trolltech went ahead and added the GNU GPL as an alternative license to Qt in September 2000, thereby completely eliminating licensing as a consideration for Qt acceptance.

By the time that the QPL was introduced, GNOME had already been developed for over a year, and the introduction of more liberal licensing for Qt arrived too late for a merger between the GNOME and KDE projects. Thus, the original tension between the pragmatist camp of the Free Software community and the "purist" liberty-focused camp resulted in two competing entrants for the category of desktop operating systems for Linux. These competing groups, however, drove the Open Source world's advancement. The practical approach of KDE, in ambitiously aiming to create an entire GUI front-end, spurred the "purists" to match KDE in creating their own desktop project. In addition, the "purists" presence in creating GNOME based entirely on open licenses encouraged the move of Qt to Open Source as well, thereby preventing a potentially dangerous compromise down the road with KDE based on a proprietary foundation.

Both KDE and GNOME are currently successful in the Linux marketplace, each competing vigorously and maintaining hundreds of developers

and applications. This multi-prong approach encourages competition in the Linux operating system sphere, as each environment strives to out-implement the other's features and interface. On the other hand, the seeming bifurcation in the Linux world is not as absolute as it might first appear: because both are pieces of Free Software based on X Windows, it is possible—and common—to run both KDE and GNOME applications at the same time, on the same machine, on the same desktop, no matter which underlying desktop environment is running. Both sets of applications appear side-by-side, and they just adopt a different visual appearance and style. Finally, because both KDE and GNOME are both Open Source and free, they are usually included with mainstream commercial distributions of Linux, providing users with the ever recurring theme of the Open Source world—choice.

While KDE and GNOME compete as desktop operating environments for Linux, they also strive for compatibility. Because they are not controlled by a single company, and want to produce the best user experience for the widest array of users, both groups have the incentive to enhance interoperability. For example, work is underway to enable GNOME applets to be embedded within KDE applications, and vice-versa; to allow drag-and-drop functional-

ity to work across multiple desktop environments; and to standardize various application layer communication protocols to allow KDE and GNOME apps to exist more seamlessly with one another.

Development of KDE and GNOME has been organized around the formation of two foundations that oversee their promotion and growth. The GNOME Foundation was created on 15 August 2000 to advance the availability and support of the GNOME desktop environment.[21] In coalition with industry leaders and corporations, including organizations such as Compaq, Eazel, the Free Software Foundation, Helix Code, HP, IBM, and Sun Microsystems, the GNOME foundation was designed to coordinate the technical direction of the overall project, and promote the overall adoption of GNOME and its underlying component and toolkit technologies. KDE initially claimed that the formation of such a foundation did not affect its future development[22]. However, KDE ended up forming the KDE League[23] on 13 November 2000. Similar to the GNOME foundation, this organization is comprised of industry leaders and companies that promote the usage

[21]For more information, please see the GNOME Foundation press release from the LinuxWorld Expo in San Jose, VA, released 15 August 2000, available online at http://www.gnome.org/pr-foundation.html

[22]For KDE's complete initial response and reasoning, please see their essay available online at http://www.kde.org/announcements/gfresponse.html.

[23]For more information, please see http://www.kdeleague.org/.

and development of KDE and Open Source desktop alternatives by corporations, governments, enterprises, and individuals. Both organizations consist of boards that are elected from the developer community in a bottom-up fashion. Thus, both desktop environments maintain a decentralized organization to coordinate marketing penetration efforts on a global scale, thereby raising awareness and usage of their particular software.

5.2.2 Technically Competitive and Innovative

Programmers for KDE or GNOME are not constrained by some of the limitations inherent in a proprietary development environment. For example, developing for KDE or GNOME eliminates the necessity for an individual programmer to "sell" his idea to the management hierarchy before time and monetary resources are allocated to pursuing that avenue of exploration. If an individual programmer has a great idea to add to an existing piece of software, or an entirely new way of doing things, he has access to the source code of the current project state, and can easily integrate his ideas into the system. If the patches are accepted by the community, these code changes are entered into the primary distribution. The lack of a developer hierarchy is a competitive advantage in the sense that the best ideas are allowed to

surface and be implemented, regardless of the "status" of the originator.[24]

While they are not constrained by some of the development limitations present in a proprietary development environment, KDE and GNOME developers lack access to the massive revenue stream that finances research and development at large, successful proprietary software companies. One of the great general advantages of all software over other areas of research, however, is that the implementation of a novel idea does not require vast amounts of capital beyond the basic computing hardware. The Free Software movement, through the GNU project, has already succeeded in creating a vast store of compilers, editors, and other tools to use in the production of software. These freely available tools make it easy for developers to begin work with a minimum of capital outlay, partially mitigating the lack of the large research and development revenue stream available to some proprietary software corporations.

There are no barriers or requirements that innovative ideas must come from large research houses. GNOME and KDE have pushed the boundaries for innovative, technically competitive operating systems. For example, KDE

[24]For a discussion of this decentralized method of harnessing complexity as it is explicitly expressed in the KDE development philosophy, please see http://www.kde.org/whatiskde/devmodel.html.

won "Innovation of the year 1998/1999" in the Software category of ZDnet's annual award at CeBIT, the world's largest computer trade show.[25] Indeed, many innovations occur at the individual level and are then cultivated and developed by the development environment of which the individual is a part.

GNOME, for example, has designed the entire system structure around a software component architecture called Bonobo[26]; each component focuses on implementing a small, specific, and correct set of interfaces and features. These components are structured around a commonly defined set of protocols that allow components written in different languages and located in different locations—even on different systems—to interact seamlessly as if they were one large, monolithic application. While common to other, early component architectures such as Apple's OpenDoc or Microsoft's ActiveX, Bonobo provides these advanced structures at the system's core, easily available for inclusion in, and extension by, additional code. Extensive use of XML, an advanced imaging model, and an implementation in efficient, portable C are further innovations that strengthen GNOME as a desktop operating system and demonstrate the potential of an Open Source development environment.

[25]Please see http://lists.openresources.com/kde/kde-announce/msg00069.html.
[26]Please see http://www.helixcode.com/ miguel/bongo-bong.html

KDE has also been at the forefront of the component-based architecture, utilizing its compound document technology known as KParts, similar in focus to GNOME's Bonobo.

The presence of two primary competing operating environments in the Open Source also leads to a rapid pace of innovation that a single hegemonic system cannot imitate. Each group of developers is looking at the other groups ideas—and, because it's Open Source, their code—and seeing the results. Each is trying out the best of the other's ideas, experimenting, and tweaking. The competition between the two groups increases the level of innovation: the competition gives each group a guide by which they can measure their own software.

5.2.3 Tech Focus: Quality Results in Code

GNOME and KDE both exhibit technical excellence in their underlying code base, in large part due to the massive peer review process by which they were developed. Algorithms and procedures in code have been developed in a competitive, transparent environment. In this environment, inefficient implementations are identified, and either lose in the competitive arena to more successful implementations, or are corrected by programmers eager to im-

prove them. Both in their underlying libraries and in their user environment applications, the GNOME and KDE environments compete openly, sharing the best ideas of the other in a cross-pollination of innovation.

Open Source operating environments such as GNOME and KDE have a much stronger security model due to the openness of their development process. The GNOME and KDE communities can tear apart the code base to ascertain any flaws in security. In addition, it is assumed that the "black hats"—those programmers who are attempting to "crack" or break security—have access to the same source code. Thus, algorithms are designed with best practices in mind, because the opposition will know exactly how the system works. Security through openness leads to the development of a widely peer-reviewed code base that has been scrutinized from a variety of perspectives by a large number of people. Security is not attempted through obscurity, or hoping that the "bad guys" do not figure out how it operates. Instead, it is secure by design. There are members within the programming community who have a particular interest in maintaining exceptional security, and hence an incentive to ensure that the code is of optimal quality. Because they have access to the source code, they can perform an audit and be positive that the

operating environment is performing securely in its intended fashion without any "hidden" backdoors or unanticipated functionality.

In addition to the general advantages for Open Source development with regards to peer review, security and privacy for Desktop Operating Environments on Open Source systems benefit from variety and choice. Instead of facing a single, monolithic operating environment to attempt to "crack," black hats face a wide collection of variation between GNOME and KDE. In addition, choice and the dispersal of a single environment spread power and control, lessening the possibility of a single corporation or organization building in features that track behavior and otherwise reduce user's privacy. Uniformity in any system favors central power, and the potential for misuse that are associated therewith. As Alexis de Tocqueville commented concerning political governments,

> "Every central government worships uniformity: uniformity relieves it from inquiring into an infinity of details, which must be attended to if rules have to be adapted to different men, instead of indiscriminately subjecting all men to the same rule."[27]

A uniform user environment makes it easier to exercise control or disrup-

[27]Please see [54]. This section available online at http://xroads.virginia.edu/~HYPER/DETOC/ch4_03.htm.

tion on a large scale. Diversity of underlying user environments lessens the chance that a single organization—whether a corporation or a government—has complete control over software implementation and security. Distribution of power combined with an open accountability of the functioning of systems as central in importance as operating environments increases both code and security quality.

5.2.4 Operator Focus: Quality Results for End-Users

Uniformity and standardization in an end-user interface provides significant advantages. New users are able to quickly learn their way around a system. Seasoned users find there is a smaller learning curve in picking up a new software application or task because much of the basic framework of the application structure is standardized and has already been learned. Training costs for companies are reduced, and productivity increases as users try new applications because the transition from the known to the unknown doesn't involve a jump to a completely different usability metaphor.

At first glance, the existence of two desktop environments in the Linux world would be a hindrance to widespread acceptance of the system to consumers. To gain acceptance in the desktop OS market, it would appear that

Linux needs a single, standardized desktop environment to compete with Microsoft Windows. After all, consistency and the resulting network effects implications is one of the reasons the Microsoft Windows operating system has risen to such high levels of consumer acceptance.

The KDE/GNOME split—against conventional wisdom—has the potential to produce greater quality results for end-users than a single desktop operating environment.[28] Indeed, a base level of interface consistency is an integral component of software quality. However, the KDE and GNOME environments already possess a certain level of consistency. Both are quite similar, using well-established variants on the windows/icons/mouse/pull-down menus (WIMP) tradition pioneered by Xerox PARC in the late 1970's and popularized by Apple Computer's Macintosh. Both respond in similar fashion to user input, and employ very similar keyboard shortcuts and visual metaphors. Indeed, someone who knows how to operate a Windows PC or a Macintosh could learn either Open Source environment very quickly.

Beyond a base level of consistency, however, eliminating alternative user interfaces eliminates competition, and thus cuts down on the rate of inno-

[28]For a more detailed discussion of these advantages, please see [55].

vation and creativity. Long term, successful consistency in user interfaces is not established from the top-down, by committee or standards making organizations. Instead, it occurs because end users express their preferences in a marketplace full of alternative choices. Successful products often have interfaces that are emulated by others in an attempt to capitalize on the skill sets already acquired by existing users. Good interface standards tend to arise from healthy competition, as users in the field express their preference for what actually works best. In an arena where there are multiple, competing interfaces, competition and variety are harnessed to evolve the user interface into a form more closely fitted to the needs of end-users. The Open Source method of using biological principles of variation and selection of the fittest in the realm of desktop user environments eventually leads to a higher quality experience for the user.

Multiple desktop environments encourage different ways of approaching the world; choice in environments thereby supports the diversity of experiences found among end users. Interfaces provide a metaphor for computing usage, thereby providing a system of enablements or constraints for particular tasks. By offering a platform with a choice of multiple desktop environments,

the Open Source world allows users to choose the environment which best represents their computing experience and particular needs. Thus, a choice among varying desktop environments that share a certain amount of consistency would attract a larger market, and provide a more quality experience to a larger variety of end-users, than a single desktop operating environment.

5.2.5 Excellence in Fully Utilizing Network Effects

Both GNOME and KDE exhibit symptoms associated with network effects. As more users install a particular operating environment, that environment becomes a target for more applications developers. As the number of applications increase, more users are drawn to the environment, creating a positive feedback cycle. Because of their free nature (free in this case in terms of initial cost) and their ability to dual-boot, it costs users very little to try the new environment. These new "experimenters" increase the total number of users, thereby potentially creating a larger market for positive feedback to take hold.

In addition to benefiting from a standard feedback loop, GNOME and KDE also can benefit from the feedback loops of other operating environments. For example, both KDE and GNOME applications can run side by

side; thus, a user, while making a choice in operating environments, is not "locked out" of the other and can continue to use applications developed for that environment. An Open Source project called "WINE"[29] is also under development for Unix-based systems that attempts to replicate the Windows API's, allowing software compiled for Windows applications to run under X Windows on Unix platforms, including within GNOME or KDE. While not complete, current versions of WINE allow users to run a variety of applications, including the common Microsoft Word and Excel 2000[30]. As WINE matures, it will provide another means by which GNOME and KDE can leverage the existing network effects present on the Windows platform.

The Open Source nature of the KDE and GNOME desktop environments also encourages the use of open protocols and API's, thereby encouraging developers to produce applications that support higher levels of interactivity. In an environment with closed protocols or "private" API's, companies without access to the hidden communication information are at a disadvantage to the company that controls that code. In an Open environment, such as in the

[29]WINE stands for "Wine Is Not an Emulator", another example of the GNU tradition of recursive acronyms. For full details of the WINE project, please see their website online at http://www.winehq.com.

[30]Screenshots and commentary available online at http://wine.godmonkey.com/.

KDE and GNOME space, other groups have equal access to the source code that underlies the API's and communication protocols, and can thereby create applications that offer more interoperability. Any software coder can look directly at the underlying code that supports the Bonobo API's in GNOME, for instance, and not only fully utilize these API's in their own software, but also directly contribute to these API's future development. As the number of nodes increase, whether in individual pieces of software on one machine or on machines across a network, the total power and value of the networked system increases exponentially.

5.2.6 Long-Term Economic Viability

Companies founded on Open Source solutions generally obtain revenue from value-added services and consulting, and feed profits back into continued development. Because of the open nature of Open Source software, however, the economic viability of a particular platform is not tied to an individual company. If a particular software system or environment has merit, it will be continued to developed regardless of the status of any specific sponsoring companies.

In the GNOME Desktop Operating Environment, for example, Eazel is

an Open Source company established by veterans of Apple Computer to make "computers easier and more enjoyable to use by developing innovative software, Internet-based services, and communities."[31] Eazel's primary software is a desktop-management and service-delivery application called Nautilus that serves as a file- and system-management application for GNOME. While tightly integrated with Eazel's sold services for network user environments, Nautilus has been released as Open Source under the GNU GPL. Its development, while heavily centered on Eazel employees, is distributed globally and would not cease were Eazel to suddenly go bankrupt or be purchased by another firm.[32]

Because of this vendor independence, GNOME and KDE have a supported life-time that is longer than any one company. As long as there continue to be users of a particular version of GNOME and KDE to supply demand, there will continue to be support in the form of coding patches and updates. This "uncoupling" of software from the control of a particular company also increases the long-term viability of support options. Because the code is open, any company or group could fill the needs of long-term support,

[31]http://www.eazel.com.

[32]Please see http://www.newsforge.com/article.pl?sid=01/03/14/0210224.

and happily occupy a market niche, even if the user base had shrunk or the "originating" company had ceased to exist.

5.3 Conclusion

The desktop operating environment is the most common point of intersection that users have with computing software. Because it forms the base upon which other applications can run and interact, it also forms the core of a network system, which operates upon the rules of network effects. These network effects have greatly magnified the influence of a first mover and have contributed to the situation that personal computers are in today: that of a market dominated by a single provider with over 90% of the market. Because of this overwhelming dominance, many users assume that the level of software quality associated with the market leader is simply the way software "is," not recognizing that there are other alternatives.

Through a close examination of Microsoft Windows, the market leader, as well as KDE and GNOME, two alternatives developed in an Open Source environment, we have evaluated each of these desktop operating environments based on our previously established criteria of software excellence. This examination demonstrated that the Open Source development model can pro-

duce software that is of comparable, or greater, quality to that produced in a closed, proprietary development environment.

6 Conclusion: Open Source's Gales of Creative Destruction

> "There's a war out there, old friend—a world war. And it's not about who's got the most bullets. It's about who controls the information. What we see and hear, how we work, what we think—it's all about the information." —*Sneakers*

This thesis has demonstrated that Open Source can produce excellent, high quality software. Development environments make a difference: they act as enablers and constraints on the evolution of products and ideas, providing incentives for certain behaviors and discouraging others. Although they are differing environments, proprietary (Closed Source) and Open Source development paradigms both provide ecosystems in which software can grow and mature. Both environments also provide the incentive structure necessary to produce quality software, as defined previously. We will now proceed to examine the success of Open Source from a complexity standpoint, witnessing how the software development environment is ideal in encouraging rapid evolution through variation and mutation. Finally, we will briefly examine the possibility of greater market success for Open Source due to its emphasis

on the liberty of code.

6.1 Open Source: Success in Greater Quality and Innovation—Variation and Mutation as Hallmarks of Evolution.

Incentives inherent in the Open Source environment point towards greater quality and innovation as compared with software produced in a proprietary environment.[1] The successes of Open Source development stem from its being able to take advantage of complexity. By utilizing a large population filled with variation and mutations, Open Source software is able to rapidly narrow the massive problem space inherent in any software design, and evolve products quickly. Closed systems, requiring a conscientious, controlled, top-down approach, are much less "organic" in their handling of the problem of software design. By modeling development on the example of evolutionary processes in complex adaptive systems in nature, Open Source development is able to take advantage of the rapid rates of improvement and innovation found in the emergent properties world of complexity theory.

The free availability of the source code in an Open Source project leads

[1]Indeed, Open Source is a disruptive technology with the potential to shake up the industry. For an overview of how disruptive new paradigms can lead to "gales of creative destruction"[56] that disturb an existing market or societal order, please see [57].

to a greater number of variants in the population of software. Modification of the source code is akin to a mutation in the biological world; the particular mutation of software is almost always an attempt at an improvement over a perceived deficiency in the original code, and thus results in a variation of the piece of software that is slightly improved over the original. A large number of these mutations are evaluated and recombined with the original software, and the best code patches are then accepted into the main evolutionary "tree" by key developers. Communication among developers is sufficiently high so that a variety of alternative solutions to a particular problem are usually put forward. In addition, the testing of alternatives in a public forum by knowledgeable programmers generally provides that the "winning" mutation is of high code quality and best meets the challenges of the problem under consideration.

According to Robert Axelrod, "variety is the engine of rapid quality improvements in an Open Source initiative."[2] According to his model of harnessing complexity, variation should be encouraged when four specific criteria are met. [3] All four of these criteria exist in the development of general pur-

[2]Please see [2], p. 56.

[3]For an application of these criteria to the Linux Operating System specifically, please see [2], pp. 56–58.

pose software, such as applications, operating systems, or desktop operating environments.

First, software exhibits problems that are long-term and widespread. Much software, and most algorithms that underlie successful software, is commonly used and addresses many of the same issues of application and communication. Benefits and improvements to the quality of these pieces of software will last for a long time and will benefit a great number of users. Algorithms demonstrate similarity across multiple problem spaces, and solutions that are optimized for a particular problem are applicable to similar problems in other domains.

In addition, software exhibits a cycle of fast, reliable feedback. Improvements are quickly recognized, and bugs in the software design or execution are rapidly brought to attention. In addition, the nature of software means that each individual programmer who is contributing a mutation to the software population pool can make the modification to a fully functioning piece of software locally, and test the modification locally. The ability to release only working pieces of software back into the general population further increases the speed of the feedback loop.

Third, software is an example of a class of problem that has a safe path for exploration. Quality software—particularly software that is maintainable over a long period of time—is designed in a modular fashion, limiting the effects of interdependence and allowing software programmers to "play" or explore without fear of catastrophe from their efforts. "Layers" in software design can be tweaked or replaced without affecting the overall system design. Thus, optimizing or adding a particular function within the greater scheme of the design is possible without disrupting other, related functionality.

Finally, many software developers fit the fourth criteria of encouraging variety: problems with a sense of imminent disaster. Many software developers see a narrow window of opportunity in which software freedom—in its philosophic sense—can be preserved in the face of stultifying controls imposed by the widespread domination of proprietary software. This ideological drive for "liberty" in software drives participants to succeed in the face of annihilation, much as the desire to survive catastrophe and radical climate change spurs a host of variations and mutations in biological species.

Software perfectly fits Axelrod's criteria for projects that would benefit from increased variety and diversity. Thus, long-term, software benefits from

a development environment that encourages an evolutionary approach to innovation and quality through a competitive, developing world. Recognizing that software is an example of a complex adaptive system is a start to the solution to the problem of software innovation and quality. By using features of such a system, including the biological concept of progressive evolutionary adaptation through mutation and selection of the fittest, software developed in such an environment will eventually outpace that designed in a more rigid ecosystem in which top-down design is superimposed on a complex underlying problem. As Eric Raymond has pointed out, "The open-source culture will triumph not because cooperation is morally right or software 'hoarding' is morally wrong, ... but simply because the closed-source world cannot win an evolutionary arms race with open-source communities that can put orders of magnitude more skilled time into a problem." [4]

6.2 Open Source: Success in Greater Market Acceptance

In addition to its long-term prospects for greater software quality and innovation due to harnessing the power of complex adaptive systems, Open Source also has the potential to obtain greater acceptance in the economic

[4]Please see [31].

marketplace due to its emphasis on freedom. Freedom in the Open Source world refers to freedom of control more than free-ness of cost.[5] This liberating aspect of freedom affords companies and individuals a variety of opportunities and advantages that are not available in a Closed Source solution.

Traditional proprietary development environments for software production do not give customers control over the product. Modifications or improvements to the software that would enable it to fit better into a particular company's existing workflow can only be added at the discretion of the vendor. In addition, companies are tied to a single vendor for future improvements and support of a previously purchased or licensed product. Closed formats and API's increase the amount of lock-in that customers have with specific vendors, and the switching costs involved often make moving to another vendor's product prohibitively expensive. The software vendor controls access to the source code upon which applications—both the vendor's and those of independent software vendors—must run, and thus control access to the software's foundation.

[5]Free-ness of cost also has a substantial impact. A. Khalak uses computational techniques to model the economic impact of introducing free goods into traditionally proprietary commercial software markets. This model is particularly useful in examining how the measured effects of specific variables result in differing outcomes for the market share and domination of different classes of software, including Open Source and proprietary. For more info, please see [58].

By utilizing Open Source software, companies gain complete control over their software. They can obtain individual components—and even entire "distributions" or collections of prepackaged software—from multiple vendor sources who compete on service, support, and distribution consistency. Or, they can obtain the original source code and customize the software to their specific needs. Much as an "open" hardware platform built around the IBM PC gave consumers control over the hardware components they utilized at the beginning of the personal computer revolution in the early 1980s, an Open Software platform allows the consumer to control the software they use.

Open Source software is inherently anti-monopolistic and serves to combat the tendency of network externalities to create monopolies in network industries. Network effects and feedback loops combined with even a small first mover advantage lead to the creation of monopolies in many software categories, including that of operating systems and desktop operating environments. With such conditions, "there is no presumption ... that superior technology wins."[6] Monopoly control relies on keeping the underlying API's

[6]Please see [59], p. 32.

and protocols secret. By "de-commoditizing"[7] the API's, a company can retain monopoly control and prevent competitors from entering a particular market. Open Source, by contrast, is anti-monopolistic. Because the source code—and hence the protocols and API's—are open and available, anyone can distribute a piece of Open Source software or create another piece of software that is also compatible. Even if a piece of Open Source software or operating system were to capture a majority of the market, no single company would be able to attain monopolistic control over the market, and competition and vendor choice would be maintained.

In addition to offering software consumers much greater control over a product that is critical to the smooth functioning of modern society, Open Source software eliminates much of the waste associated with duplicate work. While most code—over 75%[8]—is written for internal use, many problems in computer science show up in multiple different applications. Thus, components of software are subject to reuse; it hurts productivity—and company's bottom lines—when software is contracted to be written that utilizes code that has already been written—potentially multiple times—in other scenar-

[7]Please see [38].

[8]http://www.netaction.org/opensrc/oss-conclude.html#no16

ios. Open Source, publically available code would prevent "reinventing the wheel," and contribute to a much greater overall quality of code, particularly for basic or common functionality.

In the Open Source world, the Perl programming community provides an example of the benefits reaped from sharing code collections. The Comprehensive Perl Archive Network[9] was created as a common repository for modules of programming code that perform common functionalities. By having such a large collection of code publicly available, most basic functionality has already been written, and the rapid development of new applications is greatly facilitated. In addition, the shared, peer-review nature of the publicly accessible code improves the quality of these modules beyond that which might be expected were they to be designed "on the fly" by a different programmer for each new project.

Due to its anti-monopolistic tendencies as well as the economic and productivity advantages, many question whether Open Source should be encouraged as a public policy initiative. Recognizing both the economic savings and the strategic advantage of controlling their own software, many nations

[9]CPAN. Available online at http://www.cpan.org/

have begun to advocate the usage of Open Source software in their routine operations. For example, Mexico City announced on 12 March 2001 that they would be transitioning to the Open Source Operating System Linux over Microsoft Windows, saving a tremendous amount of money on licensing and enabling Mexican software engineers to gain a foothold on the city's systems.[10] Other nations, including Italy, are also considering switching to officially supporting Open Source software in their governmental offices.[11]

A public policy stance supporting Open Source software might provide a successful remedy in the antitrust proceedings against Microsoft.[12] Other remedies against Microsoft are purely punitive and retain a great deal of government oversite without necessarily encouraging innovation or attacking the root of the problem: the monopolistic tendencies prevalent because of network effects in the operating system market. A policy supporting Open Source software, by contrast, would offer a positive response that could create a viable alternative to the monopoly while also providing the other benefits of Open Source outlined above.

[10]For an interview with the Technical coordinator of Mexico City (in Spanish) with information about the reasoning behind the transition, please see [60].

[11]Please see [61].

[12]Please see [62] as well as [63].

Through its support of early software and network research, as well as its promotion of open standards for operating systems and network protocol specifications, the United States government has historically played a role in the development of "open" software. In recent years, however, the US government's public-policy support of open software and standards has waned. Many have argued for a strong government policy in favor of Open Source software[13] because of its socially beneficial features. Opponents argue that the government is not in a position to delegate software choice, and that this decision should lie in the hands of the market. Others, often known as techno-libertarians, view the government's potential involvement in this arena as problematic.[14] Regardless of one's position, however, the stance of government on Open Source—both from the position of originator and enforcer of copyright law and as a massive consumer of software—will play an increasingly major role as the disruptive force of the Open Source paradigm is experienced more widely. As Lawrence Lessig has commented,

[13]Please see [64].

[14]For a debate on the merits of public policy support of Open Source software, please see [65]. Therein, Eric Raymond argues the techno-libertarian viewpoint against government intervention, while Lawrence Lessig argues that government does have a role in enabling that the conditions for open innovation survive. Other debate participants include Nathan Newman, Jeff Taylor, and Jonathan Band.

> "Government has a role in enabling the conditions for innovation. It has a role in ensuring those conditions survive. 'We' should start thinking critically about that vital role. Or before we know it, despite the best efforts of the open-source movement to 'crush' others, others will have co-opted government to a very different end."[15]

The Open vs. Closed nature of software affects more than simply the quality of the resulting product—it affects the freedoms possible in a world built on the cyber-foundation of code. Software—and the code that undergirds it—is the architecture that is beginning to control the modern world. Code has become critically important, and thus extremely valuable. Control over the code affects the balance of power and the distribution of wealth in society. Because of the complex interrelationship between the *technology* of code, and society, the "best" development ecosystem—as defined previously in our discussion of software excellence—is not always the "winner." There are too many other variables, economic, political, and cultural, to assume that technological superiority alone determines market success. For software, information is power, and power produces positioning benefiting various interest groups. It is therefore vital that individuals not only recognize that variables in the software development process profoundly affect the structure

[15]Please see [65].

of the world, but also that choices concerning the properties of the software infrastructure of today's digital world are still being made and are not set in stone.[16] This thesis has demonstrated that, in terms of quality—which includes long-term sustainability—Open Source is a viable choice for code development environments.

6.3 Conclusion

Software development is a complex, multi-faceted ecosystem. While the traditional means of software development—a proprietary, closed-source environment—have demonstrated success in the personal computer market over the past quarter century, a new contender—Open Source—has emerged. In this thesis, we have demonstrated that not only can an Open Source development model produce quality software, but it also takes advantage of the complex adaptive systems model of diversity and adaptation to better meet the established criteria of an "ideal" software development system. Not only does the Open Source development world provide incentives for the production of "better" software, it also offers advantages of control and flexibility

[16]For a compelling, if dark, read on the power of architecture—code—to regulate, and on the narrow window of opportunity that society has in setting conditions for future liberty, please see [4].

that will encourage its gradual widespread adoption. By demonstrating that a system that allows for the free sharing and exchanging of the components that comprise its core assets can be successful and generate products that compete in terms of quality with proprietary solutions, Open Source is a harbinger of a new paradigm that may have vast consequences for the production and distribution of all information-based products.

References

[1] J. Gleick, *Chaos: Making a New Science*. United States of America: Penguin USA, 1988.

[2] R. Axelrod and M. D. Cohen, *Harnessing Complexity: Organizational Implications of a Scientific Frontier*. New York, New York: The Free Press, 2000.

[3] S. Miles, "Linux closing in on Microsoft market share," 24 July 2000. From CNET.com. Available online at http://www.canada.cnet.com/news/0-1003-200-2332817.html.

[4] L. Lessig, *Code and Other Laws of Cyberspace*. New York, New York: Basic Books, 1999.

[5] M. K. McGee, "It's Official: IT Adds Up," 17 April 2000. Available online at http://www.informationweek.com/782/productivity.htm.

[6] "Information Technology Research and Development: Information Technology for the 21st Century," 21 January 2000. The White House, Office of the Press Secretary. Available online at http://www.whitehouse.gov/WH/New/html/20000121_2.html.

[7] D. E. Knuth, *The Art of Computer Programming*. United States of America: Addison-Wesley Pub Co, 1997.

[8] G. Brahma, "Measuring Software Quality." Available online at http://www.hssworld.com/hss_mindsystem/tech_speak/software_quality.htm.

[9] T. S. Kuhn, *The Structure of Scientific Revolutions*. Chicago, Illinois: University of Chicago Press, 1996.

[10] R. Lemos, "Companies losing more to internet crime," 12 March 2001. Available online at http://news.cnet.com/news/0-1003-200-5109411.html?tag=mn_hd.

[11] T. Rauch, S. Kahler, and G. Flanagan, "Usability Techniques: What Can You Do?," *SIGCHI Bulletin*, vol. 28, pp. 63–64, Oct 1996.

[12] B. Dilger, "The Ideology of Ease," September 2000. Published by the University of Michigan Press in the Journal of Electronic Publishing. Available online at http://www.press.umich.edu/jep/06-01/dilger.html.

[13] S. Johnson, *User-Centered Technology: A Rhetorical Theory for Computers and Other Mundand Artifacts*. Albany, New York: State University of New York Press, 1998.

[14] R. A. Heavener, "The Odd Success of the US Software Industry," 4 November 1996. Available online at http://www.apl.jhu.edu/Classes/605401/hausler/cybrbob.html.

[15] J. Katz, "Geek Life," 1998. Available online at http://www.newstrolls.com/news/dev/katz/102798.htm.

[16] J. Frederic P. Brooks, *The Mythical Man-Month*. Reading, Massachusetts: Addison-Weslet, 1995.

[17] J. Markoff, "A tale of the tape from the days when Microsoft was still Micro-Soft," 18 September 2000. Available from the New York Times online at http://www.nytimes.com/2000/09/18/technology/18BASI.html.

[18] S. Levy, *Hackers: Heroes of the Computer Revolution*. United States of America: Penguin USA, 2001.

[19] L. D. Garcia, "Networks and the Evolution of Property Rights in the Global, Knowledge-Based Economy." A paper prepared for the 28th Telecommunication Policy Research Conference Alexandria, Virginia. September 2000.

[20] E. S. Raymond, *The Cathedral and the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary*. Cambridge, Massachusetts: O'Reilly, 1999.

[21] "Microsoft Annual Report for 2000," 2000. Available in print by request, or online at http://www.microsoft.com/msft/ar.htm.

[22] C. Fishman, "They Write the Right Stuff." Available online at http://www.fastcompany.com/online/06/writestuff.html.

[23] E. B. Dean, "Total Quality Control from the Perspective of Competitive Advantage," 10 June 1995. Available online at http://akao.larc.nasa.gov/dfc/tqc.html.

[24] M. Minasi, *The Software Conspiracy: Why Companies Put Out Faulty Software, How They Can Hurt You and What You Can Do About It.* United States of America: McGraw-Hill Professional Publishing, 1999.

[25] B. Pfaffenberger, "The U.S. Software Industry and Software Quality: Another Detroit in the Making?," 3 May 2000. Available online at http://www2.linuxjournal.com/articles/currents/019.html.

[26] B. Tognazzini, "If They Don't Test, Don't Hire Them," June 2000. Available online at http://www.asktog.com/columns/037TestOrElse.html.

[27] N. Randall, "Making Software Easier Through Usability Testing." Available online at http://www.zdnet.com/pcmag/pctech/content/17/17/tu1717.001.html.

[28] C. Shapiro and H. R. Varian, *Information Rules: A Strategic Guide to the Network Economy.* Boston, Massachusetts: Harvard Business School Press, 1999.

[29] R. Young, "Giving It Away: How Red Hat Software Stumbled Across a New Economic Model and Helped Improve an Industry," in *Open Sources: Voices from the Open Source Revolution*, Cambridge, Massachusetts: O'Reilly and Associates, 1999. Available online at http://www.oreilly.com/catalog/opensources/book/young.html.

[30] E. S. Raymond, "The Magic Cauldron," in *The Cathedral & The Bazaar: Musings on Linux and Open Source By An Accidental Revolutionary*, Cambridge, Massachusetts: O'Reilly and Associates, 1999. Latest version available online at at http://www.tuxedo.org/ esr/writings/magic-cauldron/.

[31] E. S. Raymond, "The Cathedral and the Bazaar." Available online at http://www.tuxedo.org/ esr/writings/cathedral-bazaar/.

[32] "Microsoft exec calls Open Source a threat to innovation," 15 February 2001. Available online at http://news.cnet.com/news/0-1003-200-4833927.html.

[33] E. S. Raymond, "How to become a hacker." Available online at at http://www.tuxedo.org/ esr/faqs/hacker-howto.html.

[34] "GNU's Not Unix! the GNU Project and the Free Software Foundation." Available online at http://www.fsf.org/home.html.

[35] C. DiBona, S. Ockman, and M. Stone, "Introduction," in *Open Sources: Voices from the Open Source Revolution*, Cambridge, Massachusetts: O'Reilly and Associates, 1999. Available online at http://www.oreilly.com/catalog/opensources/book/intro.html.

[36] R. Stallman, "Richard Stallman on the Allchin Controversy." Available online at http://weblog.mercurycenter.com/ejournal/stories/storyReader$664.

[37] B. Perens, "The Open Source Definition," in *Open Sources: Voices from the Open Source Revolution*, Cambridge, Massachusetts: O'Reilly and Associates, 1999. Available online at http://www.oreilly.com/catalog/opensources/book/perens.html.

[38] M. Corporation, "The "Halloween" Documents," 1998. An annotated memorandum. Available online at http://www.opensource.org/halloween/halloween1.html.

[39] D. Baird, "Scientific Instrument Making, Epistemology, and the Conflict between Gift and Commodity Economies." Available online at http://scholar.lib.vt.edu/ejournals/SPT/v2_n3n4html/baird.html.

[40] N. Bezroukov, "Open Source Software Development as a Special Type of Academic Research (Critique of Vulgar Raymondism)," 12 October 1999. Available online at http://www.firstmonday.dk/issues/issue4_10/bezroukov/index.html.

[41] E. S. Raymond, "Response to Nikolai Bezroukov," 19 October

1999. Available online at http://tuxedo.org/ esr/writings/response-to-bezroukov.html.

[42] P. Vixie, "Software Engineering," in *Open Sources: Voices from the Open Source Revolution*, Cambridge, Massachusetts: O'Reilly and Associates, 1999. Available online at http://www.oreilly.com/catalog/opensources/book/vixie.html.

[43] B. Schneier, *Applied Cryptography: Protocols, Algorithims, and Source Code in C, 2nd Edition*. United States of America: John Wiley & Sons, 1995.

[44] E. S. Raymond, "ESR on Quake 1 Open Source Troubles: The case of quake cheats," 27 December 1999. Available online at http://slashdot.org/articles/99/12/27/1127253.shtml.

[45] E. Levy, "Wide Open Source," 16 April 2000. Available online at http://www.securityfocus.com/commentary/19.

[46] R. Metcalfe, 30 May 1995. Remarks at the University of Virginia. Available online at http://www.americanhistory.si.edu/csr/comphist/montic/metcalfe.htm.

[47] M. Tiemann, "Future of Cygnus Solutions: An Entrepreneur's Account," in *Open Sources: Voices from the Open Source Revolution*, Cambridge, Massachusetts: O'Reilly and Associates, 1999. Available online at http://www.oreilly.com/catalog/opensources/book/tiemans.html.

[48] L. Torvalds, 25 January 2001. An interview with Linus Torvalds in searchEnterpriseLinux. Available online at http://searchenterpriselinux.techtarget.com/ searchEnterpriseLinux_Q_A_Page/0,285144,516996,00.html.

[49] "U.S. v. Microsoft: Court's Findings of Fact," 5 November 1999. Available online at http://www.usdoj.gov/atr/cases/f3800/msjudgex.htm.

[50] S. Berinato, D. Fisher, and R. Holland, "Microsoft's Outlook: Cloudy security," 12 May 2000. ZDNet eWeek:

Building the E-Business Enterprise. Available online at http://www.zdnet.com/eweek/stories/general/0,11011,2568904,00.html.

[51] M. Group, "Commentary: Microsoft co-opts open source approach," 8 March 2001. Available online at http://news.cnet.com/news/0-1003-201-5067896-0.html.

[52] J. Field, "Usability comparison: Windows Whistler vs. Gnome 1.2, KDE 2.0, Mandrake Update," 21 November 2000. Available online at http://www.newsforge.com/article.pl?sid=00/11/20/0317238.

[53] G. Moody, *Rebel Code: Linux and the Open Source Revolution*. United States of America: Perseus Press, 2001.

[54] A. D. Tocqueville, *Democracy in America*. United States of America: Harperperennial Library, 2000.

[55] B. Pfaffenberger, "Why the KDE/Gnome split is actually a good thing," 12 October 1999. Available online at http://www2.linuxjournal.com/articles/currents/010.html.

[56] J. Schumpeter, *Capitalism, Socialism and Democracy*. United States of America: Harper & Row, 1950.

[57] C. M. Christensen, *The Innovator's Dilemma*. Cambridge, Massachusetts: Harvard Business School Press, 1997.

[58] A. Khalak, "Evolutionary Model for Open Source Software: Economic Impact," July 1999. Ph.D. Workshop in GECCO-99: Genetic and Evolutionary Computing Conference, July 1999. Available online at http://web.mit.edu/asif/www/ace.html.

[59] J. Cassidy, "The Force of an Idea," *The New Yorker*, 12 January 1998.

[60] A. Bordon, "Evitar gobierno local compra de Windows," 12 March 2000. From reforma.com. Available online at http://www.reforma.com/ciudad_de_mexico/articulo/078598/.

[61] P. Willan, "Italians revolt against Microsoft supremacy," 30 October 2000. Available online at http://www2.infoworld.com/articles/hn/xml/ 00/10/30/001030hnitaly.xml%3FTemplate%3D/storypages/printarticle.html.

[62] K. Krechmer and E. Baskin, "Microsoft Anti-Trust Litigation—The Case for Standards," 2000. Property of the Standards Engineering Society. Winner of the World Standards Day paper competition for 2000. Available online at http://www.csrstds.com/WSD2000.html.

[63] N. Newman, "The Origins and Future of Open Source Software," 1999. A NetAction White Paper. Available online at http://www.netaction.org/opensrc/future/.

[64] M. Stoltz, "The Case for Government Promotion of Open Source Software," 1999. A NetAction White Paper. Available online at http://www.netaction.org/opensrc/oss-report.html.

[65] E. S. Raymond, L. Lessig, N. Newman, J. Taylor, and J. Band, "Controversy: Should Public Policy Support Open-Source Software? a roundtable discussion in response to the technology issue of The American Prospect (vol. 11 issue 10)," 2000.

Bibliography

"GNU's Not Unix! the GNU Project and the Free Software Foundation,". Available online at http://www.fsf.org/home.html.

"Information Technology Research and Development: Information Technology for the 21st Century,", 21 January 2000. The White House, Office of the Press Secretary. Available online at http://www.whitehouse.gov/WH/New/html/20000121_2.html.

"Microsoft Annual Report for 2000,", 2000. Available in print by request, or online at http://www.microsoft.com/msft/ar.htm.

"Microsoft exec calls Open Source a threat to innovation,", 15 February 2001. Available online at http://news.cnet.com/news/0-1003-200-4833927.html.

"U.S. v. Microsoft: Court's Findings of Fact,", 5 November 1999. Available online at http://www.usdoj.gov/atr/cases/f3800/msjudgex.htm.

Axelrod, R. and Cohen, M. D., *Harnessing Complexity: Organizational Implications of a Scientific Frontier*. New York, New York: The Free Press, 2000.

Baird, D., "Scientific Instrument Making, Epistemology, and the Conflict between Gift and Commodity Economies,". Available online at http://scholar.lib.vt.edu/ejournals/SPT/v2_n3n4html/baird.html.

Berinato, S., Fisher, D., and Holland, R., "Microsoft's Outlook: Cloudy security,", 12 May 2000. ZDNet eWeek: Building the E-Business Enterprise. Available online at http://www.zdnet.com/eweek/stories/general/0,11011,2568904,00.html.

Bezroukov, N., "Open Source Software Development as a Special Type of Academic Research (Critique of Vulgar Raymondism),", 12 October 1999. Available online at http://www.firstmonday.dk/issues/issue4_10/bezroukov/index.html.

Bordon, A., "Evitar gobierno local compra de Windows,", 12

March 2000. From reforma.com. Available online at http://www.reforma.com/ciudad_de_mexico/articulo/078598/.

Brahma, G., "Measuring Software Quality,". Available online at http://www.hssworld.com/hss_mindsystem/tech_speak/software_quality.htm.

Cassidy, J., "The Force of an Idea," *The New Yorker*, 12 January 1998.

Christensen, C. M., *The Innovator's Dilemma*. Cambridge, Massachusetts: Harvard Business School Press, 1997.

Corporation, M., "The "Halloween" Documents,", 1998. An annotated memorandum. Available online at http://www.opensource.org/halloween/halloween1.html.

Dean, E. B., "Total Quality Control from the Perspective of Competitive Advantage,", 10 June 1995. Available online at http://akao.larc.nasa.gov/dfc/tqc.html.

DiBona, C., Ockman, S., and Stone, M., "Introduction," in *Open Sources: Voices from the Open Source Revolution*, Cambridge, Massachusetts: O'Reilly and Associates, 1999. Available online at http://www.oreilly.com/catalog/opensources/book/intro.html.

Dilger, B., "The Ideology of Ease,", September 2000. Published by the University of Michigan Press in the Journal of Electronic Publishing. Available online at http://www.press.umich.edu/jep/06-01/dilger.html.

Field, J., "Usability comparison: Windows Whistler vs. Gnome 1.2, KDE 2.0, Mandrake Update,", 21 November 2000. Available online at http://www.newsforge.com/article.pl?sid=00/11/20/0317238.

Fishman, C., "They Write the Right Stuff,". Available online at http://www.fastcompany.com/online/06/writestuff.html.

Frederic P. Brooks, J., *The Mythical Man-Month*. Reading, Massachusetts: Addison-Weslet, 1995.

Garcia, L. D., "Networks and the Evolution of Property Rights in the Global, Knowledge-Based Economy,". A paper prepared for the 28th Telecommunication Policy Research Conference Alexandria, Virginia. September 2000.

Gleick, J., *Chaos: Making a New Science*. United States of America: Penguin USA, 1988.

Group, M., "Commentary: Microsoft co-opts open source approach,", 8 March 2001. Available online at http://news.cnet.com/news/0-1003-201-5067896-0.html.

Heavener, R. A., "The Odd Success of the US Software Industry,", 4 November 1996. Available online at http://www.apl.jhu.edu/Classes/605401/hausler/cybrbob.html.

Johnson, S., *User-Centered Technology: A Rhetorical Theory for Computers and Other Mundand Artifacts*. Albany, New York: State University of New York Press, 1998.

Katz, J., "Geek Life,", 1998. Available online at http://www.newstrolls.com/news/dev/katz/102798.htm.

Khalak, A., "Evolutionary Model for Open Source Software: Economic Impact,", July 1999. Ph.D. Workshop in GECCO-99: Genetic and Evolutionary Computing Conference, July 1999. Available online at http://web.mit.edu/asif/www/ace.html.

Knuth, D. E., *The Art of Computer Programming*. United States of America: Addison-Wesley Pub Co, 1997.

Krechmer, K. and Baskin, E., "Microsoft Anti-Trust Litigation—The Case for Standards,", 2000. Property of the Standards Engineering Society. Winner of the World Standards Day paper competition for 2000. Available online at http://www.csrstds.com/WSD2000.html.

Kuhn, T. S., *The Structure of Scientific Revolutions*. Chicago, Illinois: University of Chicago Press, 1996.

Lemos, R., "Companies losing more to internet crime,", 12 March 2001. Available online at http://news.cnet.com/news/0-1003-200-5109411.html?tag=mn_hd.

Lessig, L., *Code and Other Laws of Cyberspace*. New York, New York: Basic Books, 1999.

Levy, E., "Wide Open Source,", 16 April 2000. Available online at http://www.securityfocus.com/commentary/19.

Levy, S., *Hackers: Heroes of the Computer Revolution*. United States of America: Penguin USA, 2001.

Markoff, J., "A tale of the tape from the days when Microsoft was still Micro-Soft,", 18 September 2000. Available from the New York Times online at http://www.nytimes.com/2000/09/18/technology/18BASI.html.

McGee, M. K., "It's Official: IT Adds Up,", 17 April 2000. Available online at http://www.informationweek.com/782/productivity.htm.

Metcalfe, R., 30 May 1995. Remarks at the University of Virginia. Available online at http://www.americanhistory.si.edu/csr/comphist/montic/metcalfe.htm.

Miles, S., "Linux closing in on Microsoft market share,", 24 July 2000. From CNET.com. Available online at http://www.canada.cnet.com/news/0-1003-200-2332817.html.

Minasi, M., *The Software Conspiracy: Why Companies Put Out Faulty Software, How They Can Hurt You and What You Can Do About It*. United States of America: McGraw-Hill Professional Publishing, 1999.

Moody, G., *Rebel Code: Linux and the Open Source Revolution*. United States of America: Perseus Press, 2001.

Newman, N., "The Origins and Future of Open Source Software,", 1999. A NetAction White Paper. Available online at http://www.netaction.org/opensrc/future/.

Perens, B., "The Open Source Definition," in *Open Sources: Voices from the Open Source Revolution*, Cambridge, Massachusetts: O'Reilly and Associates, 1999. Available online at http://www.oreilly.com/catalog/opensources/book/perens.html.

Pfaffenberger, B., "The U.S. Software Industry and Software Quality: Another Detroit in the Making?,", 3 May 2000. Available online at http://www2.linuxjournal.com/articles/currents/019.html.

Pfaffenberger, B., "Why the KDE/Gnome split is actually a good thing,", 12 October 1999. Available online at http://www2.linuxjournal.com/articles/currents/010.html.

Randall, N., "Making Software Easier Through Usability Testing,". Available online at http://www.zdnet.com/pcmag/pctech/content/17/17/tu1717.001.html.

Rauch, T., Kahler, S., and Flanagan, G., "Usability Techniques: What Can You Do?," *SIGCHI Bulletin*, vol. 28, pp. 63–64, Oct 1996.

Raymond, E. S., "The Cathedral and the Bazaar,". Available online at http://www.tuxedo.org/ esr/writings/cathedral-bazaar/.

Raymond, E. S., *The Cathedral and the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary.* Cambridge, Massachusetts: O'Reilly, 1999.

Raymond, E. S., "ESR on Quake 1 Open Source Troubles: The case of quake cheats,", 27 December 1999. Available online at http://slashdot.org/articles/99/12/27/1127253.shtml.

Raymond, E. S., "How to become a hacker,". Available online at at http://www.tuxedo.org/ esr/faqs/hacker-howto.html.

Raymond, E. S., "The Magic Cauldron," in *The Cathedral & The Bazaar: Musings on Linux and Open Source By An Accidental Revolutionary*, Cambridge, Massachusetts: O'Reilly and Associates, 1999. Latest version available online at at http://www.tuxedo.org/ esr/writings/magic-cauldron/.

Raymond, E. S., "Response to Nikolai Bezroukov,", 19 October 1999. Available online at http://tuxedo.org/ esr/writings/response-to-bezroukov.html.

Raymond, E. S., Lessig, L., Newman, N., Taylor, J., and Band, J., "Controversy: Should Public Policy Support Open-Source Software? a roundtable discussion in response to the technology issue of The American Prospect (vol. 11 issue 10),", 2000.

Schneier, B., *Applied Cryptography: Protocols, Algorithims, and Source Code in C, 2nd Edition*. United States of America: John Wiley & Sons, 1995.

Schumpeter, J., *Capitalism, Socialism and Democracy*. United States of America: Harper & Row, 1950.

Shapiro, C. and Varian, H. R., *Information Rules: A Strategic Guide to the Network Economy*. Boston, Massachusetts: Harvard Business School Press, 1999.

Stallman, R., "Richard Stallman on the Allchin Controversy,". Available online at http://weblog.mercurycenter.com/ejournal/stories/storyReader$664.

Stoltz, M., "The Case for Government Promotion of Open Source Software,", 1999. A NetAction White Paper. Available online at http://www.netaction.org/opensrc/oss-report.html.

Tiemann, M., "Future of Cygnus Solutions: An Entrepreneur's Account," in *Open Sources: Voices from the Open Source Revolution*, Cambridge, Massachusetts: O'Reilly and Associates, 1999. Available online at http://www.oreilly.com/catalog/opensources/book/tiemans.html.

Tocqueville, A. D., *Democracy in America*. United States of America: Harperperennial Library, 2000.

Tognazzini, B., "If They Don't Test, Don't Hire Them,", June 2000. Available online at http://www.asktog.com/columns/037TestOrElse.html.

Torvalds, L., 25 January 2001. An interview with Linus Torvalds in searchEnterpriseLinux. Available online at http://searchenterpriselinux.techtarget.com/ searchEnterpriseLinux_Q_A_Page/0,285144,516996,00.html.

Vixie, P., "Software Engineering," in *Open Sources: Voices from the Open Source Revolution*, Cambridge, Massachusetts: O'Reilly and Associates, 1999. Available online at http://www.oreilly.com/catalog/opensources/book/vixie.html.

Willan, P., "Italians revolt against Microsoft supremacy,", 30 October 2000. Available online at http://www2.infoworld.com/articles/hn/xml/ 00/10/30/001030hnitaly.xml%3FTemplate%3D/storypages/printarticle.html.

Young, R., "Giving It Away: How Red Hat Software Stumbled Across a New Economic Model and Helped Improve an Industry," in *Open Sources: Voices from the Open Source Revolution*, Cambridge, Massachusetts: O'Reilly and Associates, 1999. Available online at http://www.oreilly.com/catalog/opensources/book/young.html.