The Open Source Alternative:

Understanding Risks and Leveraging Opportunities

Heather J. Meeker

John Wiley & Sons, Inc.

The Open Source Alternative

The Open Source Alternative:

Understanding Risks and Leveraging Opportunities

Heather J. Meeker

John Wiley & Sons, Inc.

This book is printed on acid-free paper. ♾

Published by John Wiley & Sons, Inc., Hoboken, New Jersey.
Published simultaneously in Canada.

Library of Congress Cataloging-in-Publication Data:

Meeker, Heather J., 1959-
The open source alternative : understanding risks and leveraging opportunities / Heather J. Meeker.
p. cm.
Includes index.
ISBN 978-0-470-19495-9 (cloth)
1. Open source software—Law and legislation. 2. Computer programs—Patents. 3. Copyright—Computer programs. 4. Computer software industry—Licenses. 5. Free computer software—Law and legislation. I. Title.
K1519.C6M44 2008
346.04'8—dc22

2007034466

Printed in the United States of America

10 9 8 7 6 5 4 3 2 1

Contents

Preface

In the late 1990s, working as a lawyer in a technology licensing practice, I found more and more of my clients using software provided under a very unusual license. My colleagues at the time advised clients to avoid this kind of software, because although this software was free of charge, it was likely to infringe intellectual property rights and so the risk of using it was too high. Clients, however, presented with tempting morsels of ready-to-use, tested, free software, and pressed by product release deadlines, do not tend to follow that kind of advice. So, by necessity, I began learning about this kind of free software. I had to advise clients about how to use that software, while managing rather than eliminating the risks. This is how I was thrown into the swimming pool of open source.

This book is the result of some 10 years working on open source legal issues. It is intended to be a practical guide for lawyers and businesspersons who wish to understand the legal issues surrounding open source software licensing. The law of open source is complex and constantly changing. Those called upon to make decisions about open source will find little to guide them in traditional legal materials. There is virtually no case law on many crucial open source legal issues, and the relevant copyright statutes have barely begun to account for computer software itself, much less open source.

Because this book is intended to inform both lawyers and businesspersons, it goes into only moderate detail about both legal and technical principles. There is plenty of in-depth material available on legal principles and software technology, and I have included some references to them for those who want to investigate further. The explanations of legal and technical principles in this book are intended to provide a quick summary

for those readers who are not familiar with them, and a refresher for those who are.

I have often heard advocates claim that open source will kill proprietary software, or vice versa, yet it is now plain that neither will annihilate the other any time soon. Open source and proprietary software are likely to coexist for decades to come. My hope is that this book will help you better understand the legal principles at work in the heterogeneous software landscape of today.

How to Use this Book

This book is for businesspeople, technicians, and lawyers who want to understand more about the risks and rewards of open source. The first part of this book is for everyone. The second part is for lawyers, or for others who want to find out more about the legal intricacies of open source. At the end there is a glossary, an index, and a collection of documents and forms to help you in your business.

I encourage those readers who are not lawyers to read the section for lawyers, if only to understand the complexity of the legal issues involved. That section contains more legal terminology and assumes more understanding about legal principles, but it should be accessible to non-lawyers as well.

A Note on Terminology

There is a glossary at the end of this book, but a few terms must be noted at the outset. Taxonomy is always one of the gating items in writing about open source. I have chosen terms that I hope will preserve a neutral point of view. So, in this book, the following terms have the following meanings:

"Hereditary" describes a license with so-called "viral" or "copyleft" terms, sometimes called a "free software" license. It refers to licenses such as GPL, LGPL, MPL, and CDDL. This term is not in common usage.

"Permissive" describes a so-called "non-viral" license, like BSD, MIT, or Apache. This term is in relatively common usage.

"Open source" describes any license that fits the Open Source Definition. It includes both permissive and hereditary licenses. This term is in nearly ubiquitous usage.

"Proprietary" describes a license that restricts distribution or use to object code. This term is in nearly ubiquitous usage in the open source community, and so I bow to convention in using it, even though I consider it misleading. Of course, open source is just as proprietary as "proprietary" software, because neither's copyright interests have been abandoned. The only truly nonproprietary software, in the copyright sense, is that dedicated to the public domain.

The views expressed herein are personal to the author in her individual capacity, and should not be attributed to Greenberg Traurig, LLP or any client of that firm or the author.

Heather J. Meeker
January, 2008

PART ONE

Leveraging Opportunities

It is tempting to approach using open source as a risk. But open source is first and primarily a powerful technological opportunity. Even so, participating in the open source movement is a step better taken with foresight and understanding. To many businesspeople or lawyers today, learning about open source can seem a daunting task. This part focuses on the basis and theory of open source licensing, and how it can be used in commercial businesses. It is intended to help you learn the basics of how to use open source intelligently in business.

CHAPTER 1

Introduction: How UNIX Gave Birth to Linux, and a New Software Paradigm

Lawyers and businesspeople who are first learning about open source tend to think of it as an entirely new paradigm, or a disruptive technology. But open source is easier to understand within its historical context. It is true that open source software licensing is the biggest sea change in technology licensing since software licensing began. But the more things change, the more they stay the same. This chapter outlines the historical background for the free software movement and the later open source movement, and explains why and how they arose.

In the Beginning Was the Word, and the Word Was UNIX

The term "open source" refers primarily to a type of outbound licensing paradigm, but also to a method of software development. Although media attention to both of these aspects of open source has burgeoned in the last decade, both the licensing paradigm and the development method have been in use ever since modern software was developed.

Although there are many software applications and utilities licensed under open source schemes, the "killer app" of open source is the Linux

operating system.[1] Understanding the free software movement of the 1990s without understanding UNIX is a little like trying to understand Martin Luther without knowing who the pope is—you may learn the doctrines of Protestantism, but they will seem arbitrary if you do not know their historical context. The philosophical tenets of open source can seem arbitrary out of context, and many lawyers and businesspeople struggle with them for this reason: Those who grew up in the age of Windows do not know much about an operating system, like UNIX, that they have never used.

The company that came up with the first modern computer operating system was not a software company but a telephone company. UNIX was developed by AT&T Bell Laboratories back when AT&T was a much-feared corporate monopoly. As a result, the company was operating under a consent decree from the Department of Justice that required AT&T not to engage in commercial activities outside the field of telephone service. AT&T had enormous research and development resources, and boasted among its ranks some of the best and brightest computer engineers of the day. AT&T set its engineers loose to develop technology within the corporate context of a not-for-profit subsidiary called AT&T Bell Laboratories.

In the 1970s, two scientists at Bell Labs, Ken Thompson and Dennis Ritchie, not only came up with UNIX, but invented a computer programming language in which to write it. That language was called C. The origin of the name UNIX may be apocryphal, but it was allegedly a pun on the word "eunuchs"; it is a quasi-acronym for Uniplexed Information and Computing System, and a successor to the earlier Multiplexed Information and Computing Service (MULTICS), but with more focused functionality. UNIX and C were tremendously innovative. UNIX was written to operate the computers of the day: large mainframes whose use was limited to huge corporations, such as banks and utilities, government, and academia. C was an extraordinarily flexible and powerful programming language. Many of the languages today, C++ and Java, for instance, are heavily based on the syntax of C.

[1]This is more completely called the GNU/Linux operating system, a distinction to be explained later in this chapter.

Because of the consent decree, AT&T was not allowed to exploit UNIX as a commercial product. So it licensed copies of UNIX to universities and others all over the world for one dollar. It soon became common practice for computer scientists to share their improvements and innovations for UNIX freely—no one tried to exploit the modifications, because there was no serious market for the product. Computers were still in use by relatively few organizations, all software was custom-written and had to be configured and installed individually, and there was no consumer computer industry.

Eventually, the consent decree was lifted. By this time, UNIX was in wide use by academics who were accustomed to treating it like a scholarly research project, not as a commercial product. UNIX was being used in at least two ways: to run computers to support other academic projects, such as statistical analyses and scientific calculations, and as a device for teaching students the nuts and bolts of operating system design. After the decree was lifted, AT&T started granting commercial licenses for UNIX, under original equipment manufacture (OEM)–type commercial licensing terms.[2] Once this happened, the many UNIX licensees no longer shared their modifications, which caused what is called *forking:* the development of many incompatible versions. Each vendor—IBM, Sun, and even Microsoft—developed its own "flavor" of UNIX licensed in object code form only.

The free software movement was a direct reaction to the privatization of UNIX. Computer scientists, particularly academics, thought operating systems needed to be freely available in source code form. Thus, "free software" refers to free availability of source code, not price. (As the Free Software Foundation pithily says, "Think free speech, not free beer.") This free availability was important because an operating system is a fundamental tool, and if it works improperly, slowly, or badly, all users suffer. Thus, free software became a political movement, based on a normative idea that the forking and inaccessibility of UNIX should never happen again. It is no accident that the computer science luminaries who

[2]This term is not used consistently in the industry; here I use it to mean a source code license allowing the OEM to make modifications but to distribute object code only, where the source code is designated a trade secret.

initiated the movement were men who started their careers using UNIX and struggled with the problems that its privatization engendered.

Along Comes Linux

Now, a few factors dovetailed to make Linux the vehicle of the free software movement.

In the late 1980s, computer science was undergoing a revolution. UNIX was no longer the most common operating system. This period saw the rise and fall of the first microcomputers: the Apple II, the TRS-80, and, of course, the personal computer. UNIX did not run on those boxes. There was no UNIX-like operating system for the new, cheap, Intel-type processors that were beginning to dominate the desktop market. These processors ran on DOS and later the Windows operating system, both products of Microsoft.

Several people tried to write smaller, more nimble operating systems that would be useful alternatives to UNIX. Such systems had to be compatible with UNIX programs, but free of the intellectual property rights of AT&T. Most notably, systems emerged based on the then-current UNIX interface specification. UNIX compatibility was essential so UNIX applications could run on those systems. One was a scholarly project called MINIX, written by Andrew Tanenbaum to help him teach operating systems and software at Vrije University in Amsterdam; it filled the academic void that the privatization of UNIX left behind. The other, to become far more famous, was Linux, the first version of which was written by a teenage computer programmer in Helsinki named Linus Torvalds. Torvalds released the first version of Linux in 1991.

Meanwhile, the GNU Project had developed into a major project to build a free alternative to UNIX. (GNU is a recursive acronym for "GNU's not UNIX.") By the early 1990s, this project had been in operation for many years. The mission of the GNU Project was to build an entire operating system, whereas Linux was only a kernel. An operating system includes not only a kernel, but development tools like compilers, debuggers, text editors, user interfaces, and administrative tools. The GNU Project was struggling with kernel development, and Linux arrived in time to provide the final piece of the puzzle. In tandem with the GNU Project, Richard Stallman of the GNU Project pioneered free software by developing the GNU

General Public License. The Free Software Foundation (FSF), a not-for-profit organization that supports the GNU Project, became the publisher and steward of that license. The FSF convinced Torvalds to make his kernel available under free software licensing terms, and the rest is history. The GNU/Linux operating system is what most people call Linux.

Finally, the last element of serendipity occurred: the economic recession of the early 2000s. A nosediving technology industry was desperate to cut costs and keep innovation alive, and lots of programmers were out of work and looking for something to do to keep embarrassing gaps out of their resumes. This was the perfect environment for open source to blossom. Blossom it did. Statistics on adoption of Linux are hard to come by, but most agree that it is gaining exponentially in popularity, particularly outside the United States.

Now, What Is Open Source?

To understand open source, you must understand what source code is and why access to it is important. Most computer users today use desktop computers with Windows operating systems. When you run a program on your desktop computer, such as a word processing program or an e-mail program, the file on your computer that contains the program is called, for instance, myprogram.exe. The "exe" stands for "executable." The file you are accessing is an executable file, or a file containing instructions that the computer can read and perform.

Programmers do not write executable code, they write source code. Programmers and corporate lawyers are a lot alike, in that they rarely write anything from scratch. When we write a contract, we rarely sit down to a blank page to begin. We use a model or form agreement, and add provisions from our libraries of provisions, which we have developed or collected over the years. Programmers work in exactly this way. If a programmer were writing a word processing program, he or she would need to include a way to save a file to a disk drive. But it would not be efficient—or even advisable—for the programmer to write it from scratch. Rather, he or she would use a prewritten component, or "library routine," to accomplish this. By using a prewritten routine, programmers make their coding more efficient, and they have more assurance that the

code will be free of bugs and interoperable with the platform on which the program will run.

Lawyers writing contracts reuse provisions informally and idiosyncratically, but in programming, this process is highly formal. If you are writing a program and want to use a routine to write a file to disk, for example, the development language you are using will include a library routine called something like "writefile." That routine will need some information, such as where the file will be written, the name of the file, the information to be written, and how many bytes will be required. The documentation for the "writefile" routine will specify what information needs to be communicated to the routine when it is executed. Lawyers should think of this as similar to conforming an arbitration provision in a contract with the rest of the agreement: Definitions of "parties" and "business days" should not conflict. When writing contracts, lawyers do this by making global changes to the text. Those who have never written a contract might consider instead the example of writing a slew of thank-you letters. The letter might say: "Dear_____. Thank you for the lovely_____." Part of the letter is the same every time, and other parts vary. The parts that are the same are dictated by custom and manners.

In programming, this reuse is done formally and systematically. Programmers write their program in a text processor (or development environment). The code looks like cryptic English, and comprises a series of instructions telling the computer's processor what to do. Any skilled programmer can read this code, which is called source code. But the computer cannot execute this code as it is written. Once programmers have all the code written, and have included references to the library routines they need in that code, they run a large, complex program called a compiler. The compiler translates the source code into object code—a set of binary instructions that the computer's processor can execute. A related program, called a linker, then links the resulting object code to the referenced library routines, making sure information will be passed properly between the components, and produces a program that can be executed by the computer: an "executable" file, which usually contains many object code files. It is important to understand that programmers do not necessarily need access to the source code for the library routines. They only need to know what information to send to the routines and what

information they will send back. The routines are usually "black boxes" to programmers: They do not need to know what is inside the box, just what goes in and what comes out.

To use another example that lawyers and businesspeople may recognize, a compiler is a lot like a redlining program. It runs in "batch" mode; in other words, once you enter some information (such as identifying the current and prior version and how you want the deletions to look) and press "go," the program runs without user interaction. The end product is a file that you do not edit. If you discover a mistake in the redline, you must go back to fix the document, then rerun the redlining program. Similarly, if there is a bug in a computer program, you do not edit the object or executable file. You must correct the source code and recompile the program. This is why access to source code is crucial. Without source code, you cannot fix errors, and must rely on the program vendor to do so.

And This Is Just the Beginning

Open source, then, is not exactly new. Many of its tenets and practices are almost as old as the software industry. However, open source has come to be a focus of legal discussion as a result of having transformed from a set of informal industry practices to a political movement, with attendant intellectual property licensing practices.

Open source software licensing is a very complex topic. Some legal issues related to it are quite thorny and undecided. Also, formulating best practices in open source development requires familiarity with a complex set of facts and industry practices as well as the political, business, and legal principles at work. Best practices in this area change quickly and sometimes unexpectedly. This book, therefore, is not intended to give you all the answers. Instead, it is intended to provide you with the background and tools to understand this area of law and develop your own conclusions and best practices, to better leverage opportunities and manage risk.

CHAPTER 2

Free Software and Open Source

Viruses and Freedoms

The first thing any lawyer learns about open source is that there are two kinds of open source licenses: "viral" and "nonviral."

When lawyers use the term "viral," they are referring to a license condition that requires the licensee to redistribute source code in outbound licenses only on the same terms as the inbound license. Although many find this a difficult concept, it is not fundamentally different from licensing conditions common in proprietary software licensing—a paradigm with which most licensing lawyers and businesspeople are quite familiar.

Consider, for example, the typical original equipment manufacturer (OEM) source code software distribution agreement. Such an agreement typically grants the OEM licensee the right to redistribute copies of software, subject to certain flow-down provisions that restrict the terms on which the OEM may grant end user licenses. Exhibit 2.1 shows some similarities and differences between a viral license agreement and an OEM source code license agreement.

Thus, the flow-down provisions of the GNU General Public License (GPL), in scope and structure, are not unlike those of OEM contracts that have been used in the software business for many years.

Terminology in this area is fraught with peril. The term "viral" is a pejorative term that is not well received by software developers. In addition, the term "viral" is imprecise, because sometimes it is used to refer only to the GNU General Public License and sometimes it refers to all

EXHIBIT 2.1 **GPL AND OEM LICENSES: NOTHING NEW UNDER THE SUN?**

Typical OEM Source Code Software Distribution Agreement	**Free Software License (e.g., GPL)**
License grant to copy, modify, and distribute	Same
Distribution by licensee in object code format only	Distribution by licensee must be in source code format
Flow-down provisions designate the source code as trade secret and prohibit distribution	Flow-down provisions designate the source code as open and require distribution

software licenses with "viral" terms. The imprecision arises because the GPL is considered "more viral" than other licenses (for reasons explained further in Chapter 14).

Advocates of this licensing scheme call it free software. But this term requires immediate explanation, because of the ambiguity of the word "free," which can mean either free of charge or free of restriction. It is also a term generally associated with free software advocates—those who prefer free software from a normative perspective. For those wishing to use a more neutral term, there are several alternatives, none of which is in universal parlance. The word "reciprocal" is also commonly used to describe free software licenses. Free software licenses are reciprocal in the sense that they require any licensee to "contribute back" to the community any changes the licensee distributes—though this can be misleading, for two reasons. First, this class of licenses may or may not require the source code to be made publicly available; it generally requires the programmer to make modifications available in source code form to all licensees to whom a copy of the binary form of the software is distributed. Only a few free software licenses actually require that modifications be contributed back to the original source code tree, and those licenses are not used frequently. This ambiguity is glossed over because the reciprocity is required in effect; under a free software license, the licensee cannot prevent any of its sublicensees from distributing the software—thus the software is effectively within the reach of the community. However, lawyers use the word "reciprocal" to describe provisions in agreements that place the same obligations on both

parties. Free software licenses do not do this at all; most of the conditions apply only to the licensee.

For a more accurate term, I recommend "hereditary," because the terms of the license initially applied to the software are "inherited" by all subsequent licensees of the same code, subsets of it, or variations of it. This parallels a concept in object-oriented programming that allows objects or data structures to inherit the characteristics of the parent sets to which they belong.

Regardless of the terminology, "viral" and "nonviral" software licensing regimes are quite different. Nonviral (i.e., permissive) software licenses are mostly unremarkable in the sense that they do not require any complex compliance procedures. The hereditary software licenses are what cause open source to be a topic of continuing controversy in the law and in the software industry.

Philosophy of Free Software

Chapter 1 described the technical antecedents to free software. This chapter describes the political positioning of free software in somewhat more depth. This book is a practical guide and thus treats the subject only briefly, but there is ample discussion on the Web available for those who wish to learn more.

Advocates of free software are quick to note that the use of the word "free" refers to freedom rather than price—in other words, *libre,* not *gratis*. As the Free Software Foundation (FSF) says, "Think free speech, not free beer." Most free software licenses do not prohibit the distribution of the software for a fee; however, they do prohibit the placing of restrictions on exercise of a sublicense. Thus, in practical terms, it is impossible to charge license fees for free software, for at least two reasons.

1. As an economic matter, this prohibition on restrictions makes it impossible for the initial licensor to exclude anyone from receiving a copy of the software, and therefore the licensor will not be able to extract economic rents from licensees in the form of license fees.
2. As a licensing matter, charging license fees for use of the work is generally prohibited by the actual terms of hereditary software licenses—either by expressly excluding the ability to charge license fees or by implication because charging license fees would be considered a prohibited restriction.

However, it is possible, and in fact common, to charge maintenance, customization, and other services fees for access to and modification of free software. Open source business models are based on this idea, and are discussed in greater detail in Chapter 11. Nevertheless, many feel that the rhetoric of the free software movement is antibusiness, although it is probably more accurate to describe it as anti–private property.

To understand the context for legal issues in open source, it is essential to understand the political landscape of the free software movement. By now, many people and companies have taken public positions along the political and rhetorical spectrum. A frequent complaint of those learning about open source is that what is written is usually not written from an objective point of view. (This is a particular complaint of attorneys, who are accustomed to reading treatises and outlines that state the law neutrally.) It is therefore important to understand who the leading commentators on the topic are and their position on open source.

The best-known philosophers of the free software movement are Richard Stallman, Eric Raymond, Bruce Perens, and Eben Moglen. Stallman, one of the original organizers of the GNU Project, has generally been the lightning rod for the free software movement and its most pithy and outspoken advocate. He is the author of the GPL, and remains a significant contributor to the GNU Project. Stallman believes that software should all be free and that programmers should be motivated by enhanced reputation and community good, rather than remuneration or profit. The structure of the GPL is intended to bring all software into the free software model. The hereditary nature of the license is Stallman's intentional—and ingenious—method of doing this.

For instance, Richard Stallman writes:

> Given a system of software copyright, software development is usually linked with the existence of an owner who controls the software's use. As long as this linkage exists, we are often faced with the choice of proprietary software or none. However, this linkage is not inherent or inevitable; it is a consequence of the specific social/legal policy decision that we are questioning: the decision to have owners.[1]

---

[1]Richard Stallman, "Why Software Should Be Free," www.gnu.org/philosophy/shouldbefree.html.

The essence of Stallman's philosophy is that access to software is expedient, and also a normative good.

> Software development used to be an evolutionary process, where a person would take an existing program and rewrite parts of it for one new feature, and then another person would rewrite parts to add another feature; in some cases, this continued over a period of twenty years. Meanwhile, parts of the program would be "cannibalized" to form the beginnings of other programs. The existence of owners prevents this kind of evolution, making it necessary to start from scratch when developing a program. It also prevents new practitioners from studying existing programs to learn useful techniques or even how large programs can be structured.[2]

Stallman also posits that ownership is not necessary to create incentives for people to develop software, because (1) some people will develop software for the interest and satisfaction of it with no external incentive, (2) some companies will develop software because it helps them create and use other products, such as computer hardware, and (3) software programmers can earn their livelihood providing services.[3]

Stallman is also a proponent of other political goals, some closely related to free software (such as his opposition to software patents) and some not related (such as his opposition to antidrug laws, his opposition to the Iraq war—and his exhortation to boycott Harry Potter books, due to objections about the misuse of copyright to protect them).[4] Stallman states that "intellectual property" is a propaganda term and objects to its use.[5]

Eben Moglen, who was until 2007 the General Counsel of the Free Software Foundation, is a Columbia University law professor and has been a frequent advocate, lecturer, and commentator on free software issues since his affiliation with the FSF in the early 2000s. Moglen is critical of the commercial software industry, and claims that free software will vanquish—or already has vanquished—the proprietary software industry

---

[2]Ibid.

[3]Ibid.

[4]www.stallman.org.

[5]Ibid.

due to its superior quality and robustness.[6] Moglen recently announced his departure from the board of directors and General Counsel position at the FSF to focus on his academic activities, but he will undoubtedly continue to be involved in the free software movement.[7]

Eric Raymond is a prominent software engineer, prolific writer, and open source advocate. He coined the phrase "With enough eyeballs, all bugs are shallow," which expresses the benefit of community maintenance and support that flows from the open source development model.[8] He authored the seminal article on open source development, entitled "The Cathedral and the Bazaar." In this article, Raymond explains that the development of proprietary software is like the building of a cathedral—designed by few, and built with resources provided and controlled by private means. Open source development, however, follows the model of the bazaar. In an open source development model, the best features and functionality evolve into popular use much as good ideas evolve into popular use in the marketplace of ideas. Development is a collaborative process, resources are not scarce, and no one person or organization directs the project.

Raymond has publicly aired his disagreements with Richard Stallman, including in a 1999 article entitled "Shut Up and Show Them the Code." Raymond holds libertarian political views, and his advocacy of market-driven assessment of software development models is consistent with those views. In other words, while Stallman and Moglen believe free software is a normative goal, Raymond believes it is a practical goal and means to an end—the end being better software. Eric Raymond is one of the original organizers of the Open Source Initiative (OSI). His political views on off-topic issues (such as his support of firearms)[9] have sometimes made him a

---

[6]For an example of this recurring theme in his speeches and articles, see Eben Moglen, "Anarchism Triumphant: Free Software and the Death of Copyright," http://emoglen.law.columbia.edu/my_pubs/anarchism.html.

[7]K. C. Jones, "Eben Moglen Steps Down from Free Software Foundation," *Information Week,* April 25, 2007, www.informationweek.com/software/showArticle.jhtml?articleID=199201489.

[8]This statement is generally attributed to Raymond but is also called "Linus' Law." Quoted in Moglen, "Anarchism Triumphant."

[9]www.catb.org/~esr/guns/.