Open Source for Business

A Practical Guide to

Open Source Software Licensing

by Heather Meeker

Third Edition

Third edition.

Library of Congress Control Number: 2015906061

Kindle Direct Publishing Platform, Seattle, SC

Editorial production for prior editions:
Paula L. Fleming and Doug McNair, Fleming Editorial Services
http://flemingeditorial.com

Editorial production for this edition:
Bill Bonney and Chris Lawrence, Last Mile Publishing
http://Lastmilepublishing.com

Fonts for this book are available under open source licenses.

- Roboto was developed by Google for the Android line and licensed under the Apache License.
- PT Sans was developed by Alexandra Korolkova, with Olga Umpelova and Vladimir Yefimov and licensed under the SIL Open Font License.
- Coelacanth was developed by Ben Whitmore and is licensed under the SIL Open Font License

Cover and page design:
Nat Case, INCase, LLC
http://incasellc.com

Table of Contents

Preface i

Part I: Foundation 1

1. The Philosophy of Free and Open Source Software 3
2. A Tutorial on Computer Software 15

Part II: Basic Open Source Theory and Compliance 39

3. Common Open Source Licenses 41
4. License Compatibility 63
5. Conditional Licensing 77
6. What Is Distribution? 89
7. Notice Requirements 107

Part III: Advanced Compliance 117

8. The GPL 2 Border Dispute 119
9. LGPL 2.1 Compliance 151
10. GPL 3 and Affero GPL 3 155
11. Open Source Policies 171
12. Audits and Due Diligence 175

Part IV: Intersection with Patents and Trademarks 187

13. Open Source and Patents (Grants, Defensive Termination) 189
14. Open Source and Patent Litigation Strategy 213
15. Trademarks 219

Part V: Contributions and Code Releases 225

16. Open Source Releases 227

Part VI: Additional Topics 241

17. Mergers & Acquisitions and Other Transactions 243

18. Government Regulation 255
19. Enforcement and Obstacles to Enforcement 261
20. Open Standards and Open Source 307
21. Open Hardware and Data 313
22. Recent Developments: Commercial Open Source, Source Available Licensing, and Ethos Licensing 319
E-Book, Forms, and Checklists 331
Glossary and Index 333
Case Index 341
About the Author 347

Preface

These days, open source software is everywhere. As they say, software has eaten the world, and open source has eaten software. What was once considered an odd twist on software licensing is now the bedrock of the software landscape. If you have picked up this book, you probably want to know more about how open source software licensing works and how you can use open source software in business in a responsible way that manages your risk. You may be familiar with the "killer apps" of open source: the Linux kernel, the Apache web server, the MySQL database, and PHP scripting—the so-called "LAMP stack"—as well as popular applications like the Firefox web browser or the Android smartphone platform. Open source software licensing has its own rules and customs and occupies a key position in today's world. It is the backbone of e-commerce and the go-to toolkit of developers everywhere.

This book is the result of over 20 years' work on open source legal issues in the technology business. It is intended to be a practical guide for lawyers, engineers and businesspeople who want to understand legal issues surrounding open source software licensing. But this is not a legal treatise; this book is intended to help not only lawyers but also others who want to understand what their lawyers are telling them. Technology lawyers know that technology always leads and the lawmakers and judges follow, trying to normalize the rules for new paradigms and capabilities. But in open source licensing, the disparity between settled law and best practices is extreme. That's because the first promoters of the open source licensing model—like Richard Stallman—were technologists, not lawyers; they had to invent their own rules and couldn't wait for legal culture to catch up. So, reading

about statutes and cases can help one understand open source licenses, but to understand how to make practical business decisions about using open source software, one needs to know quite a bit that isn't in the case law reporters.

As we stand on the brink of the 2020s, we have seen enforcement efforts by licensors of open source software, but also significant business successes in open source development. Today, no one in the technology world can deny that open source is here to stay, that there are real legal risks one must manage if one wants to use open source responsibly in a profit-making business, and that anyone who wishes to understand technology licensing and build a successful technology business has to understand the precepts of open source licensing. Living in the walled garden of proprietary licensing is no longer possible. Open source software started as the hardy plant growing through the cracks of that garden, and now it has taken over the landscape.

Background: UNIX, Linux, and Software Licensing

The introduction of open source software licensing is the most significant development in software licensing since software licensing began. But software licensing itself has not been with us very long, and in truth, open source licensing has been around for longer than most people realize. In fact, open source licensing was the original model for software licensing, and proprietary licensing is the newcomer. To understand how these two models developed together, we need to understand something about the history of computing.

Once Upon a Time, There Was an Operating System Called UNIX

When most people say "open source," they are referring to a set of software licenses. These licenses set the terms for use of the software by anyone who wants to use it. In this sense, open source is a licensing

model. But also, and perhaps more importantly, open source is a model for software development. The difference is crucial because the risks associated with the licensing model and the development model are very different.

The "killer app" of open source is the Linux operating system. Learning why and how Linux was developed is the best way to understand how the licensing model we currently know as "open source" was developed and how the open source model differs from the proprietary model.

Most people working in the technology business today grew up in the age of Windows, so they don't know much about another operating system—UNIX—that played a pivotal role in computing. UNIX was the reason the "free software" model came about.

In the early days of computing, UNIX was the dominant operating system. It was developed by AT&T Bell Laboratories. At the time, AT&T was such a dominant company that, as the result of an antitrust case, it was bound by a US Department of Justice consent decree prohibiting it from engaging in commercial activities outside the field of telephone service. [1] AT&T's best and brightest engineers therefore worked on computer science development via a not-for-profit entity called AT&T Bell Laboratories.

Ken Thompson and Dennis Ritchie were scientists at Bell Labs, and anyone who has studied computer programming knows their names. They were the creators of UNIX, which was the first general-purpose operating system. In the course of writing it, they invented a programming language called C. C was a flexible and powerful programming language that in today's parlance is called a "low-level" language, meaning a language that affords the programmer a high level of control over how the software interacts with hardware. C is still in

[1] "Modification of Final Judgment," August 24, 1982, filed in case 82-0192, *United States of America v. Western Electric Company, Incorporated, and American Telephone and Telegraph Company*, U.S. District Court for the District of Columbia, https://web.archive.org/web/20060827191354/members.cox.net/hwilkerson/documents/AT&T_Consent_Decree.pdf

common use today, though it has undergone many expansions and improvements—most notably C++, which is an object-oriented adaptation of C.

However, the consent decree prohibited AT&T from exploiting UNIX as a commercial product. So, in a rather extraordinary move that would set the wheels of the technology sector in motion, Bell Labs gave UNIX away in source code form under terms that allowed modification and redistribution. As they say, if you love something, set it free—and computer scientists in the 1970s and 1980s loved UNIX.

Then the consent decree was lifted, and AT&T started granting restricted licenses for UNIX under licensing terms that allowed redistribution only in object code format. This resulted in the "forking" of UNIX into many incompatible versions. Those who had previously enjoyed the ability to share modifications suddenly could not do so, and a program written for IBM's flavor of UNIX, for instance, would not necessarily run on Sun's UNIX.

The free software movement was a direct reaction to the forking of UNIX, which itself had happened because of a shift to proprietary licensing. There is nothing computer programmers hate more than a lack of interoperability—particularly at a lower layer of the technology stack, like an operating system—and some of them set out to prevent this from ever happening again.

It's important to understand that, at the time, "proprietary" licensing was not the ubiquitous practice it is today. In the 1980s, I worked on what were then called minicomputers: the PDP, DEC, Wang VS, and Quantel. Like most applications programmers of the time, I wrote custom business applications. I remember walking into an Egghead Software store one day, seeing accounting software one could buy off the shelf, and scarcely being able to believe it was possible. (A salesclerk had to talk me down, assuring me that if I bought a program for my Apple personal computer, it really would run on that computer.) I had been a programmer for years by then, and the notion of standardized software was foreign to me. Software came in one of two ways—loaded on the computer you bought from the vendor, or written by a custom software developer called a systems integrator or OEM.

And software was always delivered in source code form. Why? Before the microcomputer (or IBM PC), there was simply not enough standardization to support or require a binary-only software distribution model. Companies almost always bought their hardware and software from the same vendor, in a bundle. People doing technical support needed to use source code, and technical support was particular to the machine, operating system, and environment. (In fact, in about 1985, I met my first technical support representative and was amused by the concept because up until that time, as a programmer, I was the person doing technical support.) So why would anyone separate source code and binaries? It just wasn't sensible in the world in which I had been working. But I was about to witness a sea change, and the entire computing world was about to transform. Several years later, microcomputers were the norm, and binary software was standard. It was like all the bespoke tailors had gone out of business overnight and in their place had left rows of off-the-rack stores. And if you needed alterations, you had to violate a license to have them done.

Linux: The "Killer App"

Open source software—and particularly copyleft software licensing—might have remained a legal curiosity had it not been for Linux. Copyleft is a complicated licensing model, and the IT industry might never have invested the energy to figure it out, much less grow comfortable with it, if not for the well-deserved popularity of Linux. But if you don't know much about Linux, you are not alone; while most people have frequently used Linux, they may not have understood they were using it. Because Linux began its life as an operating system for computer scientists, it has never had a slick user interface. Therefore, many systems use Linux as an operating system core and layer interfaces like Android, Chrome, or even Mac on top of it. But if you don't know about Linux, then you need to learn: to understand open source licensing, it's essential to understand why Linux is so popular and so important.

In the late 1980s, UNIX's popularity was fading. Before the 1980s, most computers were used by large institutions: governments, educational institutions, banks, and other large businesses. The first generation of microcomputers—the Apple II; the TRS-80; and, of course, the DOS Personal Computer—changed all that in a matter of a few years. These machines ran on newer, cheaper processors. Operating systems are particular to processors, and UNIX could not easily be adapted for these platforms.

UNIX had a standard specification—a set of system routines that defined compatibility with a UNIX platform. That standard is now called POSIX. So, various people started seeking a kind of Holy Grail—an operating system that was compatible with POSIX but free of the license limitations propounded by AT&T. One such individual was a computer science professor in the Netherlands who wrote a scholarly project called MINIX. Another was a teenager in Helsinki named Linus Torvalds. Torvalds released the first version of Linux in 1991 and changed the world.

Meanwhile, Richard Stallman, a computer scientist at MIT's artificial intelligence laboratory, started the GNU Project. (GNU is a recursive acronym for "GNU's not UNIX.") The GNU Project sought to create an operating system that would be a free alternative to UNIX. A full operating system requires many elements, and the core of it is the so-called kernel—which runs the computer, manages memory, runs programs, and communicates with peripherals and other hardware. A full operating system also requires development tools like compilers, debuggers, text editors, administrative tools, and a user interface. Torvalds agreed to make his software freely available, and the Linux kernel—improved and adapted from the original version written by Torvalds—became this core of the GNU operating system. That operating system is now usually referred to as Linux.[2]

[2] Those at the GNU project remind us that it is more properly called the GNU/Linux system, but this kind of "branding" problem is endemic to open source licensing. See Chapter 15: Trademarks.

But Stallman was working in parallel on a licensing paradigm that would prevent privatization of the new, free operating system he was seeking to develop. He called this paradigm "free software," the rules of which were embodied in a license called the GNU General Public License (GPL). This license granted unfettered rights to redistribute software with the condition that source code could not be kept secret. This was the premise of copyleft—using copyright law to compel the sharing of copyrightable works of software.

True to the free software spirit, the Linux kernel has grown, changed, and improved dramatically since it was first developed. Today, it has thousands of contributors, a not-for-profit organization to maintain it, millions of adopters, and billions of users. And it is still free to change, redistribute, and improve. In effect, the open source licensing model forced an entire industry to cooperate, and the result is a robust and scalable system that is maintained by the largest players in IT, as well as by hundreds of volunteers.

But more importantly for our journey into open source software licensing, to use Linux, the industry had to use Linux's licensing terms. Therefore, the copyleft paradigm that Stallman pioneered slowly gained traction.

In about 1996, the GPL began making serious inroads into the technology world. I started practicing law in 1995, and soon after, my clients began asking about open source licenses. Most intellectual property lawyers at that time were confused and fearful when presented with questions like "Should I use this software under this license called GPL?" The easy answer was no—that's always the easy answer.

But some lawyers, like me, realized we needed to give a better answer. It was the equivalent of a client asking, "I found this quarter on the ground. Can I use it?" Some lawyers were still saying, "No, you don't know where it's been." But some of us were thinking, "That's 25 cents. Let's think about this some more." So I started trying to understand how to use open source software in a way that was consistent with the objectives of private business.

It was no help that the early free software movement promulgated a rhetoric that was decidedly anti-business. Their mission was to destroy

proprietary software, and most of my clients were proprietary software businesses. But others in the movement were focused on results rather than on doctrine, and some of my clients were early boosters of Linux. So my clients and I started traveling down a new path together, trying to understand whether we could really use free software in private business without destroying either model.

To do that, however, we had to learn a lot about this new licensing model. It was hard going. Advising on GPL is a bit like jumping into intellectual property bizarro world. There was little law on which to base our analysis, and for lawyers operating in a landscape of billion-dollar malpractice suits, there is no particular reward for dancing out on a limb to find answers. But all around us, the use of open source was burgeoning, and we had to keep up. If you want to spend a career giving safe answers, technology law is probably not the right practice for you.

More than twenty years later, I've learned a lot and am still learning—about technology; the law; and, most important, how licensing drives innovation. Whether you think open source licensing is scary, fascinating, crazy, or all of those things, this book is intended to help you navigate the world of open source to advance innovation and drive prosperity.

A personal note: Many people have asked me if I will "open source" this book, by which they mean they expect me to release it under a free content license like Creative Commons. Perhaps someday I will. But unfortunately, open source approaches are a challenge for scholarly works, where the main concern of an author would be that others would change and misrepresent the work, not that reuse would cause a commercial loss. Also, unfortunately, there are those who wholesale infringe copyright for commercial gain, and a Creative Commons release would leave me little recourse against those who do this, and who are mostly bad actors. But I wrote this book to educate and help others. If you want to use this book for an educational purpose, or translate it into another language, or send a copy to a friend or colleague, or some similar reasonable purpose, please contact me and I will do my best to accommodate your request.

Thank You

Many people helped me, directly or indirectly, with the development of the material in this book. In particular, I would like to thank Luis Villa, Alma Chao, Sabir Ibrahim, Maxim Tsotsorin, Aahit Gaba, David Pollak, and David Marr—all of whom helped me directly with the long task of writing and editing. I would also like to thank all the members of the FSFE Legal Network discussion list, and the Blue Oak Council, and all of those in the open source legal world who participate in free discussion and intellectual inquiry, for joining me on this journey that has few signposts. It is always best to walk new paths with friends.[3]

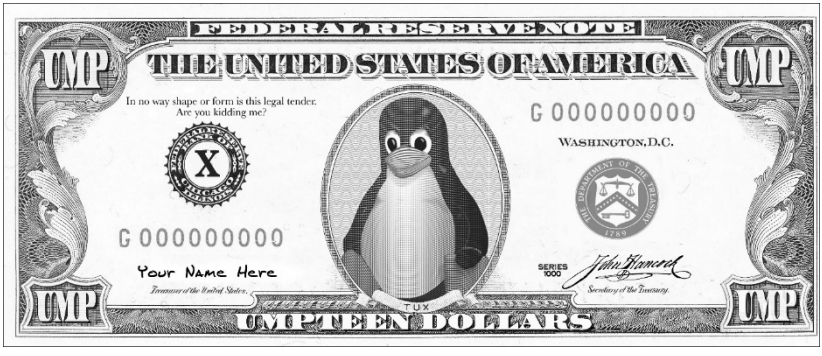
Contacting Me

If you are reading this book and need to engage an attorney to help you with open source licensing issues, please feel free to contact me. As a practicing lawyer, I am very easy to find at www.heathermeeker.com. I also welcome comments and suggestions on this book. Of course, nothing in this book should be considered a statement by my law firm or any of my clients—it only expresses my personal views—and purchasing this book is not intended to create an attorney-client relationship with you, the reader. Finally, the advice in these pages may not apply to the facts of your situation.

[3] All mistakes should be attributed to me. Only the merits of the book should be attributed to others.

Part I

A Foundation



Chapter 1

The Philosophy of Free and Open Source Software

Most people refer to software like the Linux kernel and the Apache web server as "open source" software. But the progenitors of the open source movement balk at the term *open source*—what they started was the "free software" movement. Those who are new to this subject may find nuances in terminology daunting, but these differences in terminology represent key differences in the philosophies of the participants in this movement. To make informed decisions about using open source software, it is important to understand this divide, because it bears directly on how to approach compliance programs, code contributions and releases, risk assessment, and other open source legal decisions.[1]

Richard Stallman, the progenitor of the free software movement, describes his philosophy as one promoting technical and economic freedom. As he put it, "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

1 For an interesting and informative explanation of this philosophical divide, as well as a fascinating set of interviews with some of the key technology players from the days of the Internet boom, I recommend the documentary *Revolution O/S*, directed by JTS Moore (Wilmington, DE: Wonderview Productions, 2001), DVD.

owners."[2] Stallman has called *intellectual property* a "propaganda term," and he generally objects to its use.[3]

Copyleft, therefore, is a system that uses the power of copyright law—which Stallman finds morally objectionable—to cause others to forgo copyright interests. Free software licenses are broad copyright license grants to everyone, conditioned on the requirement to allow sharing and modification of copyrightable works. This is quite different from the broader notion of open source, which includes not only free software but also software licensed under permissive terms.

In the early years of the open source movement, the anti-business rhetoric of the free software philosophy threatened to discourage open source software's adoption by business, and so a few peacemakers got together to reconcile the interests of the free software advocates and industry. This resulted in the Open Source Initiative (OSI).

Most people approaching open source for the first time think that copyleft requires people to give away software for free. That is understandable, given that the rhetoric of the free software movement tends to be anti-business and anti-capitalist. But that preconception is not exactly right, and the difference between the preconception and the reality of free software licensing is significant—allowing for the existence of successful public companies like Red Hat and GITHUB, not to mention every major consumer electronics manufacturer in existence (because of the Linux and Android platforms).

In fact, the word *free* in "free software" refers to freedom, not price—it is libre, not gratis. As the Free Software Foundation (FSF) says, "Think free speech, not free beer." Free software licenses don't prohibit you from selling copies of the software for money; in fact, that kind of noncommercial restriction is antithetical to open source licensing and, perhaps counter-intuitively, more the purview of

2 "Why Software Should be Free" by Richard M. Stallman, www.gnu.org/philosophy/shouldbefree.html.

3 "Did You Say 'Intellectual Property'? It's a Seductive Mirage" by Richard M. Stallman, www.gnu.org/philosophy/not-ipr.html. Most lawyers only object to the term being used wrongly, which leaves plenty of fodder for disapproval.

proprietary vendors like Microsoft and Adobe, who grant low-cost educational licenses with limitations on use. Nevertheless, it is impractical to base a licensing business solely on free software. The technology sector has embraced free software like Linux precisely because it does not need to make money on it. Accordingly, from a business perspective, free software is a great model for infrastructure software that everyone uses and a poor one for specialized applications. It's no coincidence that the most popular open source software is the LAMP stack for e-commerce; those who use it are making money selling goods over the Web, not selling access to the Web. The existence of free infrastructure, with no intellectual property claims to thwart it, promotes prosperity, whereas a complete lack of intellectual property would substantially thwart prosperity. The secret is finding the right balance of public and private goods to advance innovation—and not surprisingly, that is what has happened over the last 25 years as the technology business has embraced free software.

Richard Stallman is the founder of the GNU project and the free software movement. Stallman is the author of the GPL. Stallman believes that programmers should be motivated by the prospect of enhancing their reputation and doing good in the community, rather than by remuneration or profit. In that regard, of course, he differs from most programmers in private business. And if a single person embodies the antithesis of Stallman's free software thesis, it is probably Linus Torvalds—though many others might claim that role. In 1996, Linus Torvalds wrote the original Linux kernel while he was still a student at the University of Helsinki, and ever since, he has been involved in technological innovation, particularly for the Linux kernel. Today, Torvalds still provides guidance on the direction of the publicly available kernel maintained by the Linux Foundation at www.kernel.org. Torvalds is not an advocate for free software ideology so much as an advocate for freely available software. He has occasionally fallen out with those in Stallman's camp, and he is primarily a technologist rather than a political advocate.

In truth, Stallman and Torvalds represent not so much warring ideals as symbiotic ones. The licensing model could not succeed without

good software to drive it, and the software development model could not succeed without a licensing model to enable it.

The Open Source Development Model

When people talk about open source, they mean two different things: a licensing model and a development model. Most of this book discusses the licensing model, which is embodied in open source software licenses.

The open source development model is actually what creates the value of open source software. The licenses do not; they are merely vehicles to enable the development model. The best way to understand this model is through the analogy made famous by Eric Raymond in his seminal article on open source development, *The Cathedral and the Bazaar*.[4] Developing proprietary software is like building a medieval cathedral: a powerful organization (the Church) conceives a project, raises funds, and appoints a master builder, who in turn employs artisans and builders to execute the project. The progress of the project is limited by the sponsor's funds and the ability of the master builder to oversee the project. Open source development is like a bazaar. Anyone can try to sell wares. The marketplace determines what is bought and sold. Development is collaborative, resources are not scarce, and no one person or organization entirely controls the project. If the marketplace demands it, the direction of the project will change, or the project may split into multiple projects.

Many people are confused by the rhetoric of the free software movement when it begins to sound totalitarian: all software must be under GPL, no forking should happen, and so forth. This is quite different from Raymond's vision of a freewheeling market where the substance of development can change to suit the whims of the market. If it seems hard to reconcile these two visions, keep in mind that even in the freest market, there is a natural tendency to standardize. That means

[4] The online version of Raymond's article has changed over time. The current version is here: www.catb.org/~esr/writings/cathedral-bazaar/cathedral-bazaar/.

only so much forking—whether of licensing terms or software code—is useful. So even free actors in a market will voluntarily standardize their practices, or else too much effort will be wasted on reconciling incompatibilities. Open source, therefore, is a world in which freedom is paramount but some practices are discouraged—not by fiat but by consensus.

The Free Software Definition and the Open Source Definition

Given the philosophical differences between the free software purists (like Stallman) and the technological pragmatists (like Torvalds), it should be no surprise that *free software* and *open source software* are defined by their proponents in competing ways.

The Free Software Definition: The Four Essential Freedoms

0. The freedom to run the program as you wish, for any purpose
1. The freedom to study how the program works and change it so it does your computing as you wish
2. The freedom to redistribute copies so you can help your neighbor
3. The freedom to distribute copies of your modified version to others. By doing this, you can give the whole community a chance to benefit from your changes.[5]

The Open Source Definition, promulgated by OSI, covers both copyleft and permissive software licenses:

1. Free redistribution
2. Source code [must be included]

[5] Note that GNU really does number its Four Freedoms from 0 through 3. Items 1 and 3 require access to source code. See www.gnu.org/philosophy/free-sw.html.

3. Derived works [must be permitted]
4. Integrity of the author's source code
5. No discrimination against persons or groups
6. No discrimination against fields of endeavor
7. Distribution of license
8. License must not be specific to a product
9. License must not restrict other software
10. License must be technology-neutral[6]

Although the Open Source Definition is the one most frequently referred to, the Free Software Definition is easier to understand if you are first learning about this licensing model. You must first understand Freedom Zero – The freedom to run the program as you wish, for any purpose. That means no restrictions. Among technology lawyers, it means no field limitation, no territory limitation, no market limitation. As we will see later in this discussion, this essential requirement is what often separates open source licensing from other models.

OSI expressly states on its website that the GPL complies with the definition.[7] If that seems surprising, it should not. It anticipates the criticism that item 9 does not fit GPL, because GPL seeks to control software that is added to the work of authorship covered by GPL. Although these two definitions are certainly different, there is little practical consequence to their distinctions. OSI no longer approves all licenses that fit its definition, and FSF only releases licenses it considers suitable to define free software.

OSI once was a key effort to reconcile industry and free software; today, it is probably of marginal relevance—a victim of its own success. OSI was started when the war of words surrounding free software threatened to marginalize adoption of free software. The purpose of OSI is to "build bridges among different constituencies in the open-

[6] See http://opensource.org/docs/definition.php. The Debian Social Contract states that the Open Source Definition was based on the Debian definition at www.debian.org/social_contract.

[7] To understand the motivation for this comment and the answer to it, see the in-depth discussion of the GPL compliance in Chapter 8: The GPL 2 Border Dispute.

source community."[8] It is the organization responsible for certifying software licenses as open source licenses. OSI also stewards the certification trademark Open Source—although the term is probably generic and unenforceable as a trademark.[9] All of the approved open source licenses are posted on the OSI website. As of this writing, there are over 100, and several new licenses are approved every year. In the 1990s, OSI would certify almost any license agreement that fit the open source definition. However, today, OSI approves few new licenses because it wants to discourage "license proliferation"[10] and the creation of "vanity licenses"—though some might observe that many of those licenses previously approved were just that: minor variations on the major licenses that are used for only one or two projects.

In the last few years, OSI has begun to take a more aggressive posture about a few issues. For example, OSI has sent communications demanding that organizations (such as universities and SSOs) releasing software under open source licenses and simultaneously claiming patent rights reading of the software cease using the term *open source* to describe their software.[11] In 2019, the OSI declined to approve a license submitted by Mongo DB, the Server Side Public License, after lengthy controversy failed to produce a definitive answer as to whether it met the definition.

Shortly thereafter, the OSI stated that to be approved, a license must not only meet the OSD but "guarantee software freedom."[12] Richard Fontana, a noted free software advocate, stated:

I think that the business model of the license submitter can be a material consideration when assessing whether the proposed license meets the OSD and would guarantee software freedom.

[8] See http://opensource.org/.

[9] Free software advocate Bruce Perens apparently tried unsuccessfully to register the mark. See http://en.wikipedia.org/wiki/Open_source_software. For more on why trademarks become generic, see Chapter 15: Trademarks.

[10] For more on "license proliferation," see Chapter 3: Common Open Source Licenses.

[11] See the statement at https://opensource.org/node/906 for background on this initiative.

[12] https://opensource.org/approval.

Some of the controversy over these issues results from ambiguities in the Open Source Definition, which has stood unaltered for decades. However, to date, there has been no official attempt to update or clarify either the Definition or its application to such issues. The addition of the requirement to "guarantee software freedom" introduced a significant level of discretion into the approval process. That discretionary element probably existed de facto prior to the SSPL controversy but was made especially public as a result of it.

It's Not a Virus[13]

Open source advocates can be pedantic about terminology, and flame wars about using the right words are regrettably common in the open source world. But eradicating the word *viral* from discussions of open source licensing is long overdue, because this word has skewed and limited understanding of open source licensing principles. It is mostly the lawyers who are to blame. When people in the technology business, particularly lawyers, want to describe copyleft licenses, they often use the word *viral*. Unfortunately, the word is not only inflammatory but also inaccurate. For lawyers, being inflammatory is part of the game, but being inaccurate is the worst kind of sin.

There are various choices for the right term: *free software*, *copyleft*, *reciprocal*, *hereditary*. But the best choice is *copyleft*.[14] A copyleft license is one that requires, as a condition of distribution (or making available) of software binaries, that the distributor make the corresponding source code available under the same licensing terms.[15] A copyleft license "sticks" to the copyright of the software; no matter how many

[13] This section is adapted from "Open Source and the Eradication of Viruses," by Heather Meeker, March 19, 2013, http://blog.blackducksoftware.com/open-source-and-the-eradication-of-viruses/

[14] The first book I wrote on open source licensing (*The Open Source Alternative: Understanding Risks and Leveraging Opportunities*, Hoboken, NJ: John Wiley & Sons, 2008) tried to normalize the terminology for the different kinds of licenses. To my chagrin, the use of *hereditary* never caught on, so I now refer to licenses like GPL as *copyleft*.

[15] Or, in the case of licenses like AGPL, a lower threshold such as making them available via a network.

downstream distributions occur, the license terms stay the same. To a lawyer, this is like an easement on a piece of real estate (such as a public walking path over private land)—an encumbrance that runs with title, no matter how many times the property is sold. Copyleft licenses include GPL, LGPL, AGPL, MPL, EPL, and a smattering of others. But using the word *viral* to describe this concept is misleading and leads to unnecessary fears. In all the years I have been advising clients in this area of law, the single most significant misconception about copyleft licenses is due, I think, to the use of this word: *viral*.

Combining GPL code with other code in a single executable program is often referred to as creating a *derivative work*. That is because GPL says if there is any GPL code in a given program, all of the code in the program must be made available under GPL. Anything else is a violation of GPL. Companies, with visions of viruses dancing in their heads, worry that if GPL and proprietary code are combined into a single program, the GPL licensing terms will "infect" the proprietary code, and the proprietary code will be automatically relicensed under GPL. The corporate author of the proprietary code then worries that it will be compelled by law to provide the source code to its own proprietary code. This is a software company's equivalent of unleashing the big scary monster that hides under the bed. But this is simply not how copyleft works.

In fact, if a company were to combine GPL and proprietary code in a way that violated GPL, the result would be that GPL had been violated—no more, no less. What that means, legally, is that the author of the GPL code might have remedies for violation of GPL and, if the license were terminated as a result, the author would have remedies for unlicensed use of the GPL software. Both of these, essentially, are copyright infringement claims. The legal remedies for a copyright infringement claim are damages (money) and injunction (stop using the GPL code).[16]

[16] For more detail, see Chapter 5: Conditional Licensing.

There is, in actuality, no legal mechanism for GPL's infecting proprietary code and changing its licensing terms. For software to be licensed under a particular set of terms, the author has to take some action that reasonably leads a licensee to conclude that the licensor has chosen to offer the code under those terms. In contrast, combining proprietary and GPL code in a single program in violation of GPL is a license incompatibility—meaning the two sets of terms conflict and cannot be satisfied simultaneously. The better analogy for this kind of licensing incompatibility is that of a software bug rather than a software virus. Just think of the robot on the old TV show *Lost in Space*, flailing its arms and saying, "That does not compute!" and you'll get the idea.

The Philosophy of "Open"

Regardless of whether you agree with the free software philosophy, at least it is thoughtful and mostly consistent. In the last few decades, a broader notion of openness has developed, capitalizing on the success of open source software. Although these more recent and expansive notions are less rigid than the old-school free software ideology, *open* has also become one of the more overused buzzwords in technology journalism. However, it is important to understand how the notion of *open* has changed business practice in the technology industry.

There is no one definition of *open*, and many things claim to be open. But overall, open paradigms seek to involve outside parties rather than exclude them. Open models are also characterized by transparency and a focus on participation based on merit, rather than status. The licensing of intellectual property rights, whether freely or free of charge, is only part of the equation.

To bring this point home, consider the revolutionary nature of the Apple iOS platform for mobile, which, lest we forget, was released in 2007—not so long ago. This platform was called "open"—but it was not open source software. It was open in the sense that anyone could develop for the platform, using a free set of development tools. Open source advocates would be quick to point out that this was not a truly open technology environment. Developers had to sign a developer agreement,

sell only via the App Store, and follow many platform rules. But at the time, Apple's move was dramatic, revolutionary, and open in ways that few people expected.

In 2008, the Android platform was introduced. This platform expanded the notion of openness; it was based on open source software, was provided to many manufacturers, and allowed multiple app stores to compete. Meanwhile, the Linux computer O/S had been growing for quite a while. For many years, Linux did not enjoy penetration beyond the server market and embedded systems; it was notably absent from desktops. It is the closest thing to a truly open platform, but its very openness meant that it was an unattractive basis for a business model. Red Hat went public in 1999, and since then, the market for Linux distributions has thinned out. The difficulties inherent in a business model based primarily on service and support mean there is not much room for many players in the market. Yet Red Hat continues to be a thriving business, and the e-commerce economy relies on it almost exclusively.

In the 2010s, it has been very popular for industry groups to band together to support open infrastructure initiatives. Cloud Native Computing Foundation, an open source project that stewards the Kubernetes project, has achieved huge popularity. While the project was started by Google, it is now under the aegis of the Linux Foundation, which in recent years has become a corporate platform for many open source projects. It is supported by a huge roster of technology players such as Cisco, Google, VMware, Oracle, and Apple.[17]

The idea that such a constellation of technology companies would cooperate, so quickly and so willingly, on an open source project was unimaginable even 15 years ago. To the extent that there was ever truly a war of ideology between open and closed, open has won. But true to that ideology, the idea of *open* is always changing and expanding, and it doubtless will continue to deliver surprises as it helps shape the future of business.

[17] https://www.cncf.io/about/members/.

Chapter 2

A Tutorial on Computer Software

This chapter is a tutorial for those who are not computer engineers and who want to learn some of the concepts that are useful in understanding open source licensing.

What Is the "Source" of Open Source?

These days, at least according to the press, everything is open source—textbooks, yoga, seders, plant seeds, databases, and bug farm kits, to name a few. While all these things may be "open," they don't really have any source code. To understand why open source licensing is important, one must first understand what source code is.

Most computer users today use either mobile devices with iOS or Android systems or desktop/laptop computers with Windows operating systems. When you run a program (such as an *app*, or application) on your device or computer, the electronic file that composes the program is called an *executable file*. It is a file like any other on your computer, but it is in a format your computer can execute. On Windows desktop systems, these files have the filename extension *.exe*, for *executable*. (On mobile platforms, you won't see the filenames at all, but each app is an executable file.)

Computers perform complex operations by breaking the operations into very small steps that the central processor or graphics processor can perform billions of times per second. These operations include moving one to four bytes from one place in memory to another, performing simple arithmetic, and choosing the next step to execute (either by default or based on a test). Each operation is very simple, but by performing many, many operations, the computer can take actions

that humans can perceive—like printing a message on a display. The simple instructions that computers understand are in pieces too small for humans to easily understand and organize, so computer scientists have developed languages for writing computer programs that are easier for humans to process. These are what we call programming languages. Each sentence in a programming language translates into many computer instructions.

Humans express programs by writing *source code*. Computers translate source code into something that can be executed via processes called *compilation* and *interpretation*. For example, most C language programs are compiled into binary code that can be directly executed by the CPU. Programs written in Java and Microsoft's C# are translated into an intermediate binary representation called *byte code*, which is subsequently converted into executed binary by a computer program called a *virtual machine*.

Many programming languages, such as C and Java, have source code and executable forms. The source code form is not executable, meaning computers can't run it. The executable form is not readable by humans—it is essentially a bunch of ones and zeroes, sometimes called *object form*.

Source code is the language programmers use to write software. Source code looks like a natural[1] language, more or less.

Here's an example:

```
#include <stdio.h>
int main(void)
{
     int x;
     x=1;
     if (x==1) {printf ("I am the
     One.\n")};
     return 0;
}
```

[1] A natural language (as opposed to a computer language) is a language that is used by humans; English is an example.

Although this language may look strange to nonprogrammers, some of its parts of speech should be familiar. Computer languages have verbs like *include* or *print*, nouns like *x*, and adjectives like *int* (which stands for integer). Once the programmer has written the code he wants to use, he saves the file as text. He then runs a program called a *compiler*, which processes the source code text file and translates the source code into object code. Once the code is in object form, it can't be modified without going back to the source code. Although theoretically code can be "decompiled," the process is not reliable, and in any case, the decompiled code will not include the commentary that makes the source code more readable. In fact, the truth about decompilation is complex: some languages, like Java, decompile reliably, whereas others, like C, do not; and decompilers become more accurate and sophisticated every year.

When computers were slower, the time to convert a program from source code to an executable format was measurable and could slow down or interfere with humans' use of the program. Over the last 30 years (since I started programming), CPUs have gotten much faster. As processing has gotten faster, more and more operations have been moved from the main program into libraries. For example, in the days of the Apple II, each developer who wanted to write code had to move images around the screen. Today, there are many ways to animate images in a browser with a single line of source code.

Because of these advances, code is increasingly distributed in source form. For example, when your browser loads HTML and JavaScript, the browser converts the HTML and JavaScript code into executable code that displays a web page, validates input, and animates the page content. Typical web pages contain more code than could fit in the entire available memory of 30-year-old computers.

To reduce the time to transmit code over the Internet, programmers compress code to make it smaller. While this compressed code may not, technically, be an intermediate form, it looks different from the original source code. Therefore, JavaScript, which executes within a browser (see more below on this), is often changed to delete all white space. In programming, white space like blanks and tabs is usually

nonfunctional. It is only used for readability, so deleting it produces functionally equivalent source code that is just more dense to read. Therefore, the example above could be rewritten thus:

```
#include <stdio.h>
int main(void){int x;x=1;if(x==1){printf("I
am the One.\n")};return 0;}
```

This is the same as the example above but with most of the white space removed. For scripting languages like JavaScript, removing the white space reduces the number of characters in the source code and can therefore speed up the downloading of the code to the user's browser, where it executes. For programmers who are trying to write complex web applications that must be delivered via small pipes (like your mobile device), every bit of speed helps.

Building, Linking, and Packaging

In reality, most code is far more complex than in our example above. In the real world of programming, our little example would not be very useful. However, small bits of code can be quite useful if they are stitched together properly. For instance, a code snippet that adds numbers or finds a square root can be essential. But a program in the real world might need to use that bit of code over and over, and the programmer only wants to write it once. That is where building and packaging come in.

Of course, few programmers today are keen to write a routine to find square roots because others have already written such routines for them. So, there are techniques for building programs in the real world that allow programmers to reuse existing routines. An existing square root routine has already been tested and therefore is more reliable to use than one written from scratch. These existing code routines are often called *libraries*. Libraries are analogous to definitions in a contract. Just as a lawyer will refer to a defined term rather than reproduce the entire definition at every reference point, a programmer will use a library routine to perform a well-defined, common operation.

When a programmer builds a program, he first writes the code in source code form and compiles it into object code form. Then he uses a program called a *linker* to link his object with the objects of library routines. If he does this, he does not need the source code for the libraries. A well-written library is usually a *black box* to the programmer, meaning the programmer does not need to know what is inside the box but just what goes in and what comes out—the information required to stitch it all together. This information is called an *interface definition* or sometimes an API (application program interface).[2]

Once the objects are all stitched together, they are called an executable program. The *executable program* has all the pieces it needs to run.

Let us now suppose that the programmer finds a bug in the library routine or maybe just a use case that does not work for the code he has written. For instance, suppose the library selects a date on the Julian calendar (the one used in the United States) but the programmer wants to select a date on the lunar calendar. The library routine will not work for the programmer's purpose, so the programmer will need to access the source code to improve it. This is because compiling is a one-way street—it's not possible to change compiled code. If the programmer wants to make a change to the code, he has to go back to the source code, make the change, recompile the code, and relink the program.

For the lawyers reading this book, think about a redlining program. A redlining program runs in *batch mode*—once the user initiates the program, it runs without user interaction. If you want to change the redline, you go back to the original file, change it, and rerun the batch-redlining program. Compiling source code works the same way, and this is why access to source code is so important.

You can't fix bugs without source code. You can't make changes or improvements without source code. All you can do with object code

[2] People use the term *API* to mean lots of things, but this usage (respectfully submitted) is the most accurate meaning. Using API to mean code libraries is inaccurate but, unfortunately, rather common.

is build it into a larger program, and all you can do with executables is run them.

JavaScript

This language deserves its own section because it is crucial to modern web development and because it causes much confusion among nonprogrammers, for several reasons. First, JavaScript is not Java. Java is a programming language—a compiled one—that is very popular in web deployment. JavaScript, on the other hand, is a scripting language—with no compiled form—that runs programs inside a web browser. Imagine that you are ordering a pair of shoes from an e-commerce seller and filling out the shipping address. The code that checks your input, forces you to complete required fields, and otherwise directs how you interact with the web page is probably JavaScript.

The important thing to understand about JavaScript is that, like HTML, JavaScript is delivered to the user's browser in source code form and executes locally on the user's computer. This characteristic makes a big difference in open source licensing.

PERL, Python, PHP, and Other Scripting Languages

There are other popular scripting languages commonly in use, such as PERL, Python, and PHP. *A scripting language* is a high-level language, meaning a small amount of its code can accomplish a lot. Like JavaScript, these scripting languages execute in source code form, but they are often executed on the "back end" rather than in the user's browser. Scripting languages usually run on top of an *interpreter*, virtual machine, or language engine. To run the script, you need to have the interpreter installed on the user's system. The interpreter knows how to process the script.

Layers of Computing

Contemporary computer processing happens in many layers. In this respect, computing has changed enormously over the last few decades. Once, a programmer wrote a single program that ran on a single processor. In those days, every bit of functionality took up scarce time and space in the computer's processor. Today, processors are bigger and faster and can handle many programs at once, so computing has become more modular.

A computer system today, such as the one on your desktop, usually looks like Figure 2.1.

The operating system is the "traffic cop" of the computer. It tells which programs to run, keeps track of which ones are running, assigns priority to them, and mediates between the programs and the real world (such as keyboards, screens, and printers).

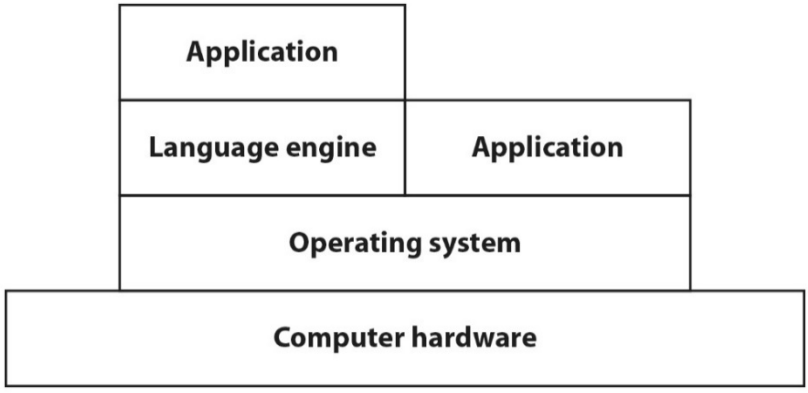


Figure 2.1 Computer system architecture

Some aspects of open source software licensing depend on how these elements interact. Let's take a more granular look, using Figure 2.2. How these elements communicate can be important. For example, you can see that only the operating system "talks" to the hardware. This is a common approach because the means of communication—what hardware pathways communicate with what hardware—may vary from one physical computer to another. Therefore, the operating system

must be engineered to work on a particular hardware platform. However, let us suppose that the developer of Application 2 does not want to write a different version of his application for every computer. Instead, he wants to write as few versions as possible while still reaching a large computing audience. He accomplishes this by using a concept called *abstraction*, which is central to modern computing. If the designer of the operating system (such as Windows, Linux, or iOS) provides a specification or API for using that system, then the developer only has to write the program once for that platform. To do this, the operating system vendor uses a set of standard system calls—or an API for that operating system. For example, that API might include an API for showing graphics on a display screen, sending a file to a printer, or getting input from a keyboard. If the programmer follows the syntax and rules for that API, his program will run on any operating system using that API.

This method of creating layers of abstraction promotes not only standardization but also security. An application programmer should not be able to do just any old thing—such as overwrite memory used by other programs or send instructions to the display to draw figures that won't fit there. Limiting the scope and permissions under which programmers operate is sometimes called *encapsulation*. It defines what it means for a program to be compatible with a particular operating system, and it sets limitations on what the program can do without causing technical problems.

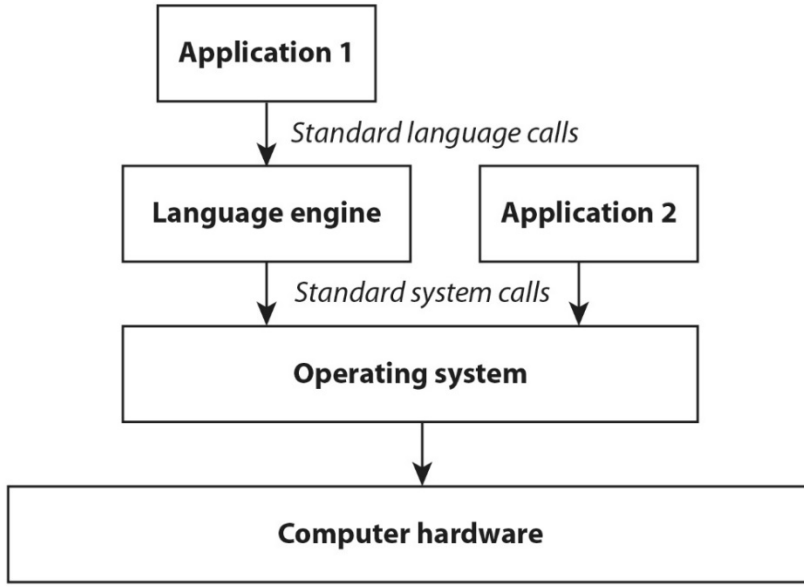


Figure 2.2 Interaction of the elements of a computer system

It's important to understand that these layers are fundamentally arbitrary—they can be set or changed by the operating system provider. But unless they are used by everyone, standardization will fail. So in fact, APIs for standard computing platforms like operating systems are slow to change. For example, Linux APIs derive from UNIX APIs that were developed in the 1970s at Bell Labs, Android runs on Linux, and iOS runs on a different UNIX derivative called BSD—but the APIs available to Linux, Android, and iOS developers are substantially similar and have evolved from a common root that is decades old.

You probably noticed that Figure 2.2 contains an application that runs on top of a language platform instead of directly on the operating system. This further layer of abstraction does two things. First, if the API for the language is consistent across many operating systems, this consistency provides even more standardization. The mantra for the Java language, for instance, is "Write once, run anywhere." Java is a very popular language platform. You are probably using it without knowing

it, because it's on most desktop and laptop computers and runs many web-based programs. When we get to the details of complying with copyleft licenses like GPL, these concepts will be crucial. To understand modern computing, one needs to think in terms of horizontal layers. Each layer has its own means of abstraction. In fact, sometimes there is an additional layer to consider, as shown in Figure 2.3.

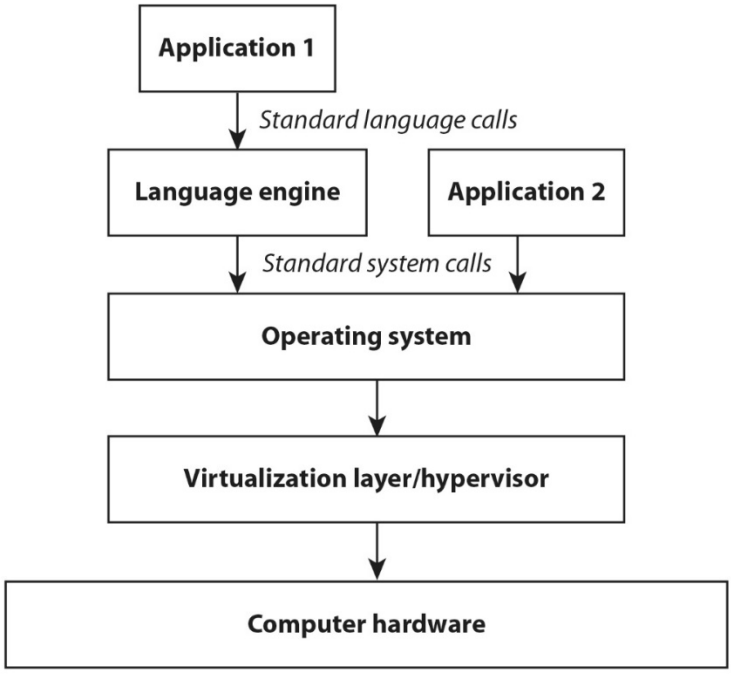


Figure 2.3 Interaction of the elements of a computer system, with virtualization layer

The virtualization layer allows a user to run programs written for one operating system on a computer that uses another operating system. In fact, this picture can get quite complicated, with multiple operating systems on one computer. But the main thing to understand is that with abstraction, very complex and standardized computing is possible.

What Is an Operating System?

When we are dealing with open source software, Linux is often the operating system component in Figure 2.3. As mentioned previously, an operating system is like the traffic cop of the computer—on a multitasking system like most general-purpose computers today, it juggles the running of multiple programs and the allocation of hardware resources, making sure each has its own memory space, initiating and terminating programs, and swapping between tasks by picking up the threads of processing where they last left off. The operating system also interfaces with computer hardware, such as keyboards, displays, printers, and modems.

The part of the operating system that does all this is always running while the computer is on. In Linux systems, it is called the *operating system kernel*. This is to distinguish it from other elements of an operating system distribution, or *distro*. To use an operating system kernel on a computer, you also need other programs: the *bootstrap* (the low-level program that launches the operating system kernel when you turn the computer on), compilers and linkers (to create executables for that particular operating system, from source code), and a user interface. Over time, the number of Linux distros has condensed. Today, the most important ones are Fedora (Red Hat) Linux, Ubuntu Linux, and Debian. There are also other specialized distros for embedded systems, real-time applications, or other specific purposes.

For the Linux operating system, these system tools consist mostly of the GNU project of the Free Software Foundation. The FSF is therefore very particular about referring to the operation system as "GNU/Linux"—though most people simply refer to it as Linux.

When a programmer writes an application program, he only wants to write it once. The operating system helps with this by abstracting the information necessary to run a program on a given physical computer. In other words, if you want to write a program for Windows, iOS, or Linux, there is a set of protocols that tell you how to display something to the user on a screen, accept keyboard entries, print documents, access the modem, and so forth. This set of protocols is called the *specification*

for the operating system. Often, this abstraction is accomplished by libraries of software called *standard system libraries*. For instance, if you want to write a file to a disk for a Linux system in the C++ programming language, you would invoke a standard library to do that. The specification tells you what information you need to pass to the standard library and how to invoke the standard library. In Linux, the standard libraries are sometimes referred to as SYSCALL—code that is stored in a library of that name and that enables standard system calls. In other words, if you write a program that uses SYSCALL, you have written a program that works with Linux. The specification, or *standard interface*, for an operating system defines what that operating system is.

As you read in the introduction, Linux began as a reimplementation of UNIX. There is a standardized specification for UNIX called POSIX (Portable Operating System Interface[3]), maintained by IEEE's Portable Application Standards Committee. About 80% of Linux SYSCALL functions are identical to corresponding functions in POSIX. So, although LINUX started as an implementation of UNIX, it has grown beyond that phase. There are standard functions in LINUX that do not exist in UNIX. The important element to understand here is that many of the standard system calls predate LINUX by quite a bit and therefore predate GPL.

What Is an Application?

Applications are the computer programs you are most familiar with—email programs, word processors, drawing programs, mobile apps, and so forth. They are programs that interact directly with users. In a certain abstract sense, there is no difference between an operating system, a language engine, and an application. They are all programs, they are all written in source code, and they all run on your computer. The difference is how they interact with each other.

[3] The X is an addition that follows the custom of naming "flavors," or versions, of UNIX with an acronym that ends in X.

The processing of an application takes place within a portion of computer memory called *application space* or *user space*. Remember that memory is the area in your computer's processor where a program is running—memory is not a disk or USB drive (which is *storage*). Your computer segregates its memory into different *name spaces*, or areas. Remember that the operating system is like a traffic cop, telling applications where to run and with what priority. The application program has an inherently limited set of functions available to it, meaning it can only do what the operating system allows it to do.

So, what constitutes an application and what constitutes application space are arbitrary in a large sense but well defined for a particular operating system platform. The separation between operating system space and application space can break down for smaller, embedded systems. But as computing power gets faster and cheaper, more and more systems are implemented in a way that separates the two. This issue is blurred when applications (like Gmail) run in the browser. When an application runs in a browser, is the application the browser (e.g., Chrome or Firefox) or is the application Gmail? This conundrum affects the interpretation of some open source licenses.

Another way to look at the separation between operating system space and application space focuses on security. Suppose you are in a retail business, like a restaurant. When you walk in, you see places to sit and waiters whose job it is to serve food to you. But there is another part of the restaurant, in the back, where food is prepared, and you can't walk into that part unless you have a key. In the kitchen, other things happen—deliveries of groceries, office work like accounting and staff scheduling, and so forth. If the waiter wants to make a food order, he passes a ticket through to the kitchen, via a hole in the wall. No key is required, but only orders are passed through and only food comes out. The designer of the restaurant has determined, arbitrarily, where the dining room ends and the kitchen begins. As a diner, you only have access to the dining area, and in that area, you can only do certain things. You can't see the grocery deliveries or the staff roster, and what's more, you don't need to. You are there to eat dinner.

The kitchen is like an operating system, the dining room is like application space, the diners are like applications, and the waiters are like standard system calls. The grocery deliveries are the hardware, which resides outside the system. Applications can only run in a specific area of the computer's memory; diners are limited to the dining room. Applications can only make requests to the operating system via standard system calls; the diners get their meals via the waiters, who have a prescribed method of submitting them to the operating system. The kitchen staff prepares meals according to the cooks' culinary skills and the diners' orders so that the meals arrive on time given the diner's orders. The operating system interacts with applications, sets the priority of services to them, and determines how the applications' requests will be fulfilled. The operating system interacts with the hardware; the kitchen staff takes grocery deliveries and puts out the trash.

You, as a diner, have limited information about how the restaurant runs. But the average diner doesn't want to know how the restaurant runs; he only wants dinner. The kitchen staff is trying to manage the dining experience and doesn't want diners running through the kitchen, which could be dangerous and distracting to the kitchen staff.

Therefore, an application is a program that has limited access to hardware, which must interact with the operating systems in a prescribed way, and which must run within the space allocated by the operating system. As a result, an application is a simpler and more efficient to program.

Dynamic Linking and Static Linking

It would be possible, in theory, to sit down with a blank page and write a program that consisted of a single source code file. But no one writes software that way. Lawyers should understand this well because they never write complex documents from scratch; instead, they start with a model or example and tailor it to their needs. Some lawyers who find they are making the same changes every time start to create true form documents that can be more efficiently prepared by filling in the

information that changes from one situation to the next. If you are not a lawyer, perhaps you have prepared a form letter or a presentation and then started to prepare another one that you thought was similar. You probably used some portions of the original letter or presentation as a model but discarded other portions.

Programmers write their programs in the same way, but they take this approach of systematic reuse to its logical extreme. Doing so is not plagiarism or cheating—it's good coding practice. That is because code is functional; it has to work correctly. Any code that has already been tested properly should be reused—if it is efficient to do that—rather than rewritten. Programmers, like lawyers, use terms of art both in source code and in the human documents that describe the execution of the program. Unifying those terms of art means that more programmers can be added to a team with minimal learning costs.

For example, suppose the programmer wants the user to input his telephone number into a program. Telephone numbers have a set format—in the United States, one example of the format would be 1-617-542-5942, where 1 is the country code, 617 is the area code, 542 is the exchange, and 5942 is the number. There are different formats for different countries and cities. So, the programmer wants to use some code to confirm that the number is in the right format before allowing the user to move on.

Clearly, many people have had to do this simple computing task, but the programmer does not know all the possible valid formats. So it makes sense for the programmer to use an existing library routine to do it. The programmer will send the input from the user (a string of characters typed in on the keyboard) and will expect to receive in return a Boolean value: TRUE for OK, FALSE for not OK. If such a library routine is well written, all the programmer should have to know is what information to send to it and what the returned information means. The working of the routine is a black box to the programmer, as explained above. To extend the restaurant analogy, if you trust that the kitchen is well run, you don't need to know what happens there—you only need to give the input (your order) and receive the output (your meal).

Now consider that the programmer wants to do many of these tasks. For instance, if the user is filling in a page of information, every field may need to be checked. Programmers use libraries of code for lots of standard and contained computing purposes—parsing user input, sending output to a printer, and doing mathematical calculations, as well as even more complex tasks like sending an email, checking whether the modem is on, or checking for program updates. In none of these cases does the programmer want to reinvent the wheel. The programmer wants to focus on what new things his program should be doing.

Here is an example of the function described above:[4]

```
BOOL ValidatePhone(CString Num)
     {
     BOOL  RtnVal = TRUE;
     if(Num.GetLength() != 11)
     {
           RtnVal = FALSE;
     }
     else
     {
           int   Pos;
           int   NumChars;
           NumChars = Num.GetLength();
           Num.MakeUpper();
           for(Pos = 0; Pos < NumChars; Pos
++)
           {
                 if(!isdigit(Num[Pos]))
                 {

     if(!isspace(Num[Pos]))
                                 RtnVal =
FALSE;
                 }
           }
     }
```

[4] This function comes from Code Project: www.codeproject.com/Articles/1874/Validating-E-Mails-amp-Phone-Number/.

```
return RtnVal;
}
```

In this example, the value returned is a *BOOL*, or a Boolean true or false. The input is a string (of characters) called Num. This routine only checks that the number of characters is correct and that all the characters are digits or spaces. The first line of the routine above shows what information needs to be passed to it (a string) and what is returned (a Boolean value). If the programmer knows this routine will accomplish his goal, all he needs to know is this first line. When he wants this routine to execute, he includes a line like this:

```
until ValidatePhone (MyNum). . .{[code to
input MyNum from the user]};
```

This line will execute until the number is valid and the routine returns a true value.

Once the programmer has all the routines he wants to use, and once he has compiled all the routines into object code, then he needs to stitch them together. In other words, when he writes the line above, the computer needs to find the object for the above routine and execute it, then pass the answer back to the program. To do this, the computer needs to know where to find the routine and where to send the answer back to. This process might look like Figure 2.4.

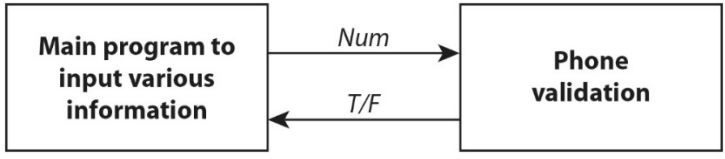


Figure 2.4 Communication between a program and routine

The entire program will contain both these objects plus information about the entry point from the main program to the phone number routine.

Even assuming the program has efficiently separated routines into modules, there are more decisions to make to create a program that

executes efficiently. Programming always involves a trade-off between memory use and speed. For example, the entire program could exist in one big binary blob and would run very quickly, but then it would use a lot of memory.[5] To achieve the right balance of speed and memory, a programmer creates a software *build*.

Consider the diagram in Figure 2.5, which shows the build of a program that uses a date verification routine.

As you can see, the Linking option requires less memory. Think of the memory used as the amount of space taken up by the boxes and diamonds. In the Linking option, only one copy of the Date OK routine is used, whereas in the Inline Function option, it appears twice, even though it is identical. But the Linking option will execute more slowly because instead of processing by automatically falling down to the Date OK function, the computer must find the function (represented by the dotted lines above), execute it, and return processing to the main line of processing.

In some languages, like C++, there are different ways to link. Have you ever used a computer and seen the error message "DLL Not Found"? In this case, your computer was instructed to execute a dynamically linked library but could not find it in memory. When a routine is called via *dynamic linking*, the computer must find the routine at execution time, execute it, then flush it from memory and return to the program that called it. The implication is that until the moment the routine runs, it does not need to be present. If, by mistake, the routine is not there and the program never calls it, nothing goes wrong. With *static linking*, by contrast, the routine is always in memory and is part of the same binary file as the routines that call it. That means there cannot be such an error; the routine is by definition resident in memory.

[5] Keep in mind that memory is the area where a computer holds and executes programs; storage is where one saves files, such as on hard disks, CD-ROMs, or USB drives.

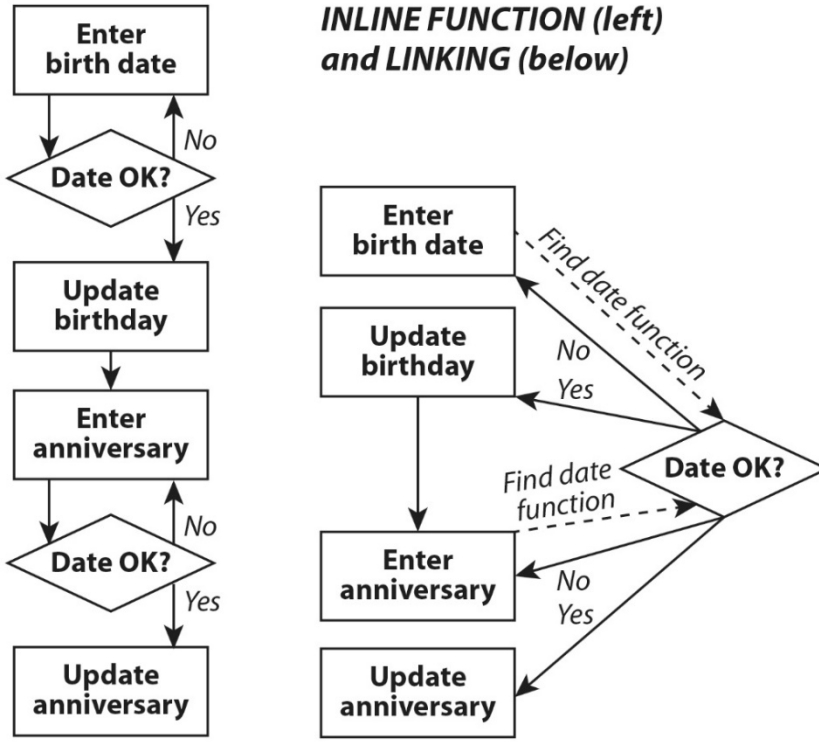


Figure 2.5 Build of a program that verifies a date

Static linking executes more quickly than dynamic linking. The time it takes for the computer to find, execute, and flush the dynamically linked routine may be very small, but if the computer must do this many times or if the speed of processing is crucial (such as in real-time applications like audiovisual streaming, gameplay, or monitoring operations), dynamic linking can be an unworkable choice. However, static linking takes up more memory because all of the routines must be in memory at once. Static linking can also deteriorate launch times because all of the routines must load into memory when the program starts.

These build techniques are readily interchangeable. The same code can be built in various ways, simply by instructing the build program how to do the build. This is done by a set of commands called a "build

script," which contains commands like "Build all of these routines as dynamically linked," or "Build all of these routines as inline," or "Pick and choose for different routines." But modern computer compilers and linkers may not always do what you tell them to do; some have features that will *optimize* the build of a program by choosing the build method based on efficiency.

Monoliths and Loadable Kernel Modules

Some operating systems are *monolithic*, meaning that the operating system consists of one big binary. Linux is a *quasi-monolithic* architecture. Most of the elements of a Linux kernel are statically linked and loaded at run time.[6] However, some elements—particularly hardware device drivers—can be implemented as dynamically linked modules. This fact underlies a very important legal question discussed in greater depth in Chapter 8: The GPL 2 Border Dispute.

Header Files

A header file describes the information that must be passed from one binary file to another so that the two can interoperate. In the date verification routine ("Date OK?"), above, the input might be a date like June 26, 1906, and the output data might be a binary number (0 for OK, 1 for not OK). A header file might look like this:

```
extern int dateOK(int mm, int dd, int
yyyy);
```

The presence of the word `extern` in the code means that the routine can be used by routines in other source files; this tells the computer that the information in the interface needs to persist long enough to be captured by the other file. The type `int` means that the function will return a single integer value. In computing, an integer is a

[6] *Linux Internals*, Section 2.15, by Peter Chubb and Etienne Le Sueur, www.cse.unsw.edu.au/~cs9242/11/lectures/08-linux_internals.pdf.

whole number, positive or negative. In our routine, this return value will be 1 or 0.[7] That is the output of the function. The function takes three input variables: `mm`, `dd`, and `yyyy`.[8] These input values are sometimes called *arguments*. In our example, each of these variables is of an integer type. `DateOK` is the name of the function; when the calling program wants to invoke this function, it will use a statement like this:

```
if (dateOK(6,26,1906)) [update record] else
[go back to input];
```

The bracketed items represent other programming statements that we need not consider for now, but they follow the logical flow of the diagrams above.

The date verification routine will contain much more processing. For example, it will check that the month is between 1 and 12, that the year is in an expected range, and that the day is within the number of days for the given month. But here is the important point: we don't need to know any of this detail to use the routine. All we need to know is what the function does, the fact that it has been tested and works, and what return value to expect. The header file contains only the information that is strictly necessary to use the function: the name of the function, the number, the type and order of arguments, and the return type. Accordingly, the function is not software code, per se. It is a definition of an interface. (This fact, important in the discussion of the nature of derivative works in software law, is discussed in more depth in Chapter 8: The GPL 2 Border Dispute.)

Containers

The sudden increase in popularity of container computing over recent years has introduced a new technical paradigm that has implications for open source licensing. *Containers*, pioneered by the company Docker, are a new approach to virtualization.

[7] We could have used a Boolean return value, of course. An integer value is used for simplicity.
[8] We could have used a date type value, of course. But that would be no fun.

Virtualization has been around for a long time and is used by most sophisticated IT systems. Virtualization allows a system manager to run a guest operating system on top of a host operating system. Thus, for example, you can run a Linux kernel on top of a Windows computer system and thereby run Linux programs on Windows.

In web computing, system operators want to be able to do this quickly—extremely quickly—as different users want to run different programs on the system. So system operators want the flexibility to create virtualized systems instantaneously. They also want to be able to stop using a virtualized guest when it is no longer needed, freeing those computing resources for other uses.

In computing, this idea is sometimes referred to as "pets or cattle." Virtualized systems, historically, have been "pets" that require care and feeding. The system operator creates one and, if something goes wrong, tries to fix it and keep it going. That's because generating a new system involves enough overhead in terms of time and computing resources that fixing the existing system is more attractive. The system operator would prefer a system that never needed to be fixed—if something went wrong, he would just kill it and bring up a new one to replace it. But that would only work, of course, if the new *instance* (a single copy of the virtualized system) could be created extremely quickly. "Cattle" refers to this situation, in which an instance is indistinguishable from its substitute and thus can be generated or killed as needed.

That metaphor is a bit brutal, so I have an alternative: garden and landscape. If the plants in your garden turn yellow, you try to adjust the water or soil to make them healthy again. In commercial landscaping, sick plants are just replaced with new ones, and the result looks just like the original landscape.

To understand how containers work, think about a shipping container, which is standardized and fungible. If you want to be able to change containers easily, they need to contain everything necessary for their own operation.

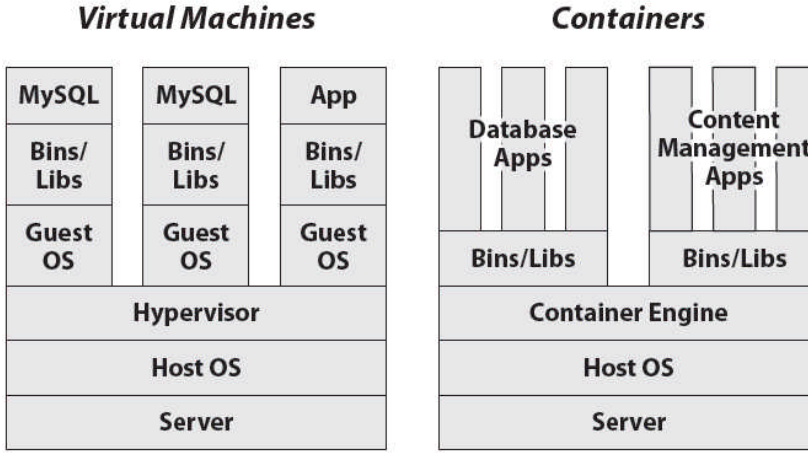


Figure 2.6 Traditional versus container virtual computing

Figure 2.6 shows a rough comparison of virtualization vs. containers. On the right, the "bins/libs" are the system files that would otherwise be part of a guest operating system. But instead of being shared by multiple applications, each comes "containerized," with all the necessary files in one package. This means it is easier to landscape instead of garden.

What difference does this make to open source licensing? Virtualized systems are usually built from the ground up (from source code or individual objects) by the system operator. But the operator can grab prebuilt containers from the Web. This means the system operator may not know what software is inside the container. Why should he care? After all, isn't it just cattle? As you will see in this book's material on open source compliance, it's impossible to comply with the notice requirements of open source licenses—much less any other requirement—if you don't know what is in your product. A container may "suck in" many software components without a useful bill of materials to say what it contains. Also, it is the nature of containers that determining what is inside the final container can be very difficult, unless you have built it from scratch.

So, containers are a hot new technology, but they also present information problems that can make open source compliance difficult. I've heard it said that grabbing prebuilt containers from the Web and using them is irresponsible for technical and security reasons, and I leave that judgment to others. However, it is certain that using prebuilt containers can lead to massive open source noncompliance. At this point, the industry is catching up, and the more responsible purveyors of containers are starting to provide meta-information about what components are included and the licensing notices for them. But for the time being, containers have played havoc with open source licensing requirements.

Part II

Basic Open Source Theory and Compliance

Chapter 3

Common Open Source Licenses

The Open Source Initiative has approved over 100 open source licenses. Some people point to this as a fault with open source licensing—which is odd, because there are many more proprietary licenses, one for each product. In fact, only a handful of the more than 100 approved OSI licenses are in wide use, and many of the others are simply variations on that handful.

Here are the licenses you need to understand:

- GPL
- LGPL
- Mozilla/Eclipse
- BSD
- MIT
- Apache 2.0
- AGPL

Open source licenses fall into two categories: permissive and copyleft. *Permissive licenses* are very simple: they allow you to do whatever you want with the software as long as you abide by notice requirements. Notice requirements are not complicated to comply with, but for binary distributions, they can be an administrative challenge to implement. (For details on how to do notices, see Chapter 7: Notice Requirements). Like all open source licenses, they provide the software as-is, with no warranties. So permissive licenses can be summarized as follows:

- Do whatever you want with the code.
- Use at your own risk.
- Acknowledge me.

There are many permissive licenses, and for some popular licenses such as BSD and MIT, there are hundreds of variations on them. But because they are permissive, they all generally work the same way. For a list of permissive licenses, take a look at the list created by Blue Oak Council at www.blueoakcouncil.org.

Copyleft is more challenging to understand. In a nutshell, copyleft says, in addition to the above:

- If you make binaries available, you must make the source code for those binaries available.
- The source code must be available under the same copyleft terms under which you got the code.
- You cannot place additional restrictions on the exercise of the license.

Copyleft falls into several categories:

- Ultra-strong (AGPL)
- Strong (GPL)
- Weaker (LGPL)
- Weak (Eclipse, MPL, CDDL)

The strength has to do with the scope of surrounding software that may be subject to the copyleft requirements. GPL is the broadest; it requires that any program that contains GPL code must only contain GPL code. The others draw the line more narrowly: LGPL allows dynamic linking to other code, and the weak copyleft licenses like Eclipse or MPL allow any kind of integration, as long as the Eclipse or MPL code is in its own file.

AGPL is called ultra-strong, but not because of its scope, which is the same as that of GPL. Instead, it is called ultra-strong because AGPL

applies its copyleft conditions (access to source code) in some situations where the code is not distributed (mainly software-as-a-service).

Dissecting the Open Source License

Open source licenses are conditional copyright licenses. (For a detailed analysis of why this matters and the legal implications for enforcement, see Chapter 5: Conditional Licensing.) All open source licenses grant all the rights of copyright, with no field restrictions or limitations. They do impose conditions on exercise of the license, but if you follow the conditions, you are not limited as to the type of use, location of use, number of copies, and so forth as you might be for a proprietary license. To put this in perspective, consider that there is an entire industry that helps companies audit whether they are within the scope of their proprietary licenses. For example, there are tools that allow IT managers to limit the number of users for a company or the number of computers on which a piece of software is installed. None of this is necessary with open source—all you have to do is abide by the conditions. Compared to proprietary software licenses, open source licenses are actually quite easy to comply with.

An open source license contains no obligations for the licensor. The licensor grants licenses but does not promise to do anything. Also, technically, the license does not contain any obligations for the licensee, only conditions to exercise the license.

Patent License Grants

Some open source licenses contain provisions about patents. The first open source licenses were written back when software patents weren't as prevalent as they are today. Software patents and business method patents (which often claim inventions practiced via software systems) became much more popular in the United States in the 1990s, after some seminal cases that confirmed they contained patentable subject matter.

Open source licenses, for the most part, contain two kinds of patent provisions: licenses and defensive termination provisions.

The license grant in certain licenses (such as Apache 2.0, Eclipse, Mozilla, and GPL 3.0) runs with the license to the software. If a contributor to the code has a patent on the code, that contributor grants a patent license to all recipients of the code to enable them to exercise the open source license.

In addition, almost all open source licenses that contain express patent licenses also have defensive termination provisions. If the licensee asserts a patent, the licensee can lose rights under the open source license.

These provisions are discussed in detail in Chapter 13: Open Source and Patents. However, it is worth noting here that the licensee in an open source license does not grant any patent rights—only the licensor does. There are no grant-backs or cross-licenses in open source because the license is not a contract, does not have an acceptance mechanism, and contains no obligations for the licensee. It contains conditions—one of which may be a defensive termination provision—but no obligations.

Direct Licensing

Open source licenses are direct licensing models—they do not grant any right to grant further sublicenses. Instead, they allow redistribution of modified or unmodified code. When an author releases code under an open source license, the grant of rights is automatically made to every recipient, regardless of how or when the code is received. Figure 3.1 shows how the grant of rights flows, regardless of whether the author of modifications delivered the modified software to the recipient.

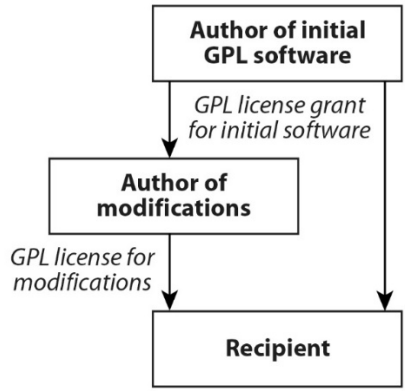


Figure 3.1 Direct licensing model of open source

The direct licensing model has a few corollaries. If a distributor violates an open source license, then although the distributor may lose his rights, downstream recipients do not. This is because the grant of rights never flowed from the distributor in the first place. Unless the downstream recipient also violates the license, the grant of rights is unaffected.

Moreover, open source licenses are never transferred. GPL version 3 is explicit about this, saying:

> An "entity transaction" is a transaction transferring control of an organization, or substantially all assets of one, or subdividing an organization, or merging organizations. If propagation of a covered work results from an entity transaction, each party to that transaction who receives a copy of the work also receives whatever licenses to the work the party's predecessor in interest had or could give under the previous paragraph, plus a right to possession of the Corresponding Source of the work from the predecessor in interest, if the predecessor has it or can get it with reasonable efforts.

But once you understand the direct licensing model, this is obvious. If a company (Buyer) purchases the assets of another company (Target), and Target delivers to Buyer the open source code Target is

using, then after the transaction is done, Buyer is exercising its own license to the code, direct from the author. The license never needs to be transferred because it was granted to all recipients (including Target and Buyer) in the first place. Proprietary licenses, in contrast, must be transferred from one licensee to another—which can be a challenge in mergers and acquisitions (M&A) transactions.

Common Open Source Licenses

Table 3.1 lists the most common open source licenses.

License		Comments
Affero GPL 3.0	*Yes*	*Ultra-strong copyleft, like GPL but source code requirements are triggered by SaaS use.*
Apache Software License (1.1)	*No*	*Apache 1.0 is largely no longer in use; version 1.1 removed the "advertising" clause.*
Apache License, 2.0	*No*	*Permissive license but far more detailed terms than BSD, MIT, or Apache 1.0 or 1.1; contains express patent grants.*
Artistic license	*No (though the point has been debated)*[1]	*Not copyleft, but has more restrictions than most permissive licenses. Many projects under this license are dual licensed under GPL.*
(New) BSD license	*No*	*Template license—many variants are in use. Major variants are the "3-clause" and "2-clause" variants. Earlier versions contained an advertising clause.*
Boost License	*No*	*Mostly used for the Boost project as a baseline license; however, some elements of the Boost project do not use it.*
Common Development and Distribution License (CDDL)	*Yes*	*Based on MPL. Successor to Sun Public License.*

[1] http://en.wikipedia.org/wiki/Artistic_License.

License		Comments
Common Public License 1.0	*Yes*	*Successor to IBM Public License. See also Eclipse license.*
Eclipse Public License	*Yes*	*Successor to CPL*
GNU General Public License (GPL) version 2	*Yes*	*Most commonly used license. Strong copyleft. Applies to Linux kernel.*
Mozilla Public License 1.1 (MPL)	*Yes*	*Weak copyleft. Applied to the Firefox browser.*
MySQL (GPL + FLOSS Exception)	*Yes*	*Like GPL, but allows linking to open source code.*
Mozilla Public License 2.0 (MPL)	*Yes*	*Weak copyleft. Applies to the Firefox browser.*
OpenSSL/SSLeay	*No*	*Permissive licenses, deprecated, used only for the OpenSSL project. OpenSSLeay contains this provision: "The license and distribution terms for any publicly available version or derivative of this code cannot be changed." However, it is generally understood to be a permissive license.*
Sun Industry Standards Source License (SISSL)	*Yes*	*Deprecated; now largely superseded by CDDL. For more information, see* www.openoffice.org/FAQs/license-change.html.
W3C License	*No*	*Permissive. Note that W3C is a standards organization; this license covers copyrightable material, and standards can cover other types of intellectual property.*
zlib/libpng license	*No*	*Permissive license*

Table 3.1 Most Common Open Source Licenses

GPL

The GPL is the prototypical *free software* or copyleft license. It is generally considered the most widely used open source license.[2]

GPL Versions

Version 1.0 is no longer in use. Version 2.0 was released in 1991, and version 3.0 was released in 2007.

Versioning of licenses can be confusing. While a small number of projects are released under a single version of GPL (such as GPL version 2 only, which is used for the Linux kernel), most are released under a given version and any subsequent version. A recipient taking code under a given version of GPL (such as version 2) has the option to use the code under that version of the license (version 2) or any subsequent version (currently versions 2 or 3).

The choice to release under a version and any later version betokens great faith in the license steward. If the steward issues a subsequent version that is less favorable to a licensor, the licensor's rights may be compromised. However, this issue is handled with an interpretational sleight of hand—while the recipient can use the license under any version, the licensor is never held to any license grants it did not make. This is probably most meaningful for GPL version 2, which contains no express patent license. Theoretically, if an author released code under GPL version 2, it is not held to the patent grants in version 3. The theory, however, is that no subsequent version truly adds grants of rights, only conditions.

Reading GPL Version 2

Most people who read GPL version 2 complain to me that it is hard or even impossible to understand. In truth, its language is not difficult to understand, but its organization does not help. Of course, there is a

[2] There are many ways to measure use, of course. GPL is applied by the most projects (as measured by various sources), but a measurement weighted by use would doubtless skew more toward Apache 2.0, MIT, or BSD.

big difference in language between GPL and conventional software licenses. GPL version 2 was an attempt at plain-language drafting—something that many lawyers, particularly in the technology practice, favor.

The first portion of the license is a preamble, roughly equivalent to recitals in a conventional license agreement. According to the law governing interpretation of contracts, this portion would probably be considered interpretive background but not part of the license terms.

The first paragraph of the GPL is numbered 0 instead of 1. In computer programming languages like C, ordinal counting begins with 0. For example, the first element of an array is 0—a convention that causes many novice coders to stay up all night debugging code.

The main terms of GPL are divided into three parts: distribution of unaltered source code, distribution of unaltered binaries, and distribution of altered code. This is probably what makes the license confusing to those who are accustomed to conventional licenses, which are almost never written this way.

Several provisions of GPL 2 have nicknames, such as these:

- Section 5 is the "No Contract" provision that states that the GPL is a license and not a contract.
- Section 7 is the "Liberty or Death" provision, which states that if the recipient may not distribute the code under the terms of GPL without other restrictions, the recipient may not distribute the code at all. This provision would come into play if there were conflicts between the licensing terms for GPL and proprietary code, for instance, or between GPL and patent licenses or nondisclosure agreements.

The GPL does not have a governing law provision. This is not an omission—it is an attempt to "internationalize" the license and thereby prevent its forking into national versions. While most lawyers consider any choice of law provision better than none, the drafters of GPL placed harmonization with local law above certainty when making this decision. Accordingly, background law will dictate what state's or country's law will be used to interpret the GPL in a particular instance.

"Special Exceptions"

Generally, GPL only comes in its official form, but there are a few variations of GPL. Each of these has been developed ad hoc and takes the form of a *special exception*, or a set of additional permissions, granted by the licensor. Each of these exceptions weakens the scope of GPL. The most popular are listed in Table 3.2.

These exceptions can be idiosyncratic, so if you are dealing with a variation of GPL that applies an exception, you must read it every time and analyze it every time.

Exception	Used For	Meaning
GCC Runtime Library Exception **www.gnu.org/licenses/gcc-exception-3.1.html**	C runtime libraries for the GNU C compiler	Broad exception that removes all requirements of GPL for use of runtime libraries used via any "Eligible Compilation Process."
Classpath Exception **www.gnu.org/software/classpath/license.html**	GNU Classpath project (reimplementation of Java libraries) and OpenJDK	Allows linking to proprietary code. Note that this allows any kind of link, but classpath files are likely to be linked dynamically.
FOSS/FLOSS Exception **www.mysql.com/about/legal/licensing/foss-exception/**	MySQL application interface	Allows linking GPL code to other open source code (including code under permissive licenses). This exception has been revised over time, most recently in 2012.

Table 3.2 Special Exceptions to GPL

Other than the FOSS exception, these exceptions are practical measures to ensure that the licensing of runtime libraries needed by a development tool (like GCC) or a language engine (like Java) does not require the corresponding application to be licensed under GPL. In other words, you can develop proprietary applications with GCC or to run on a Java platform. Note that these exceptions are more permissive

than LGPL, which is primarily used to enable the library to be used as a dynamically linked library for an application (such as a plug-in).

The Lesser General Public License (LGPL)

LGPL is *lesser* than GPL because it has a narrower scope. It is sometimes called the *Library* GPL because it is suitable for use with free software libraries. The preamble to the license says, "We use this license for certain libraries in order to permit linking those libraries into non-free programs." The FSF expressly discourages the use of LGPL because "it does [l]ess to protect the user's freedom than the ordinary General Public License." The FSF believes that some libraries should be available for proprietary development and some should not. "[W]hen a library provides a significant unique capability, like GNU Readline, that's a horse of a different color. The Readline library implements input editing and history for interactive programs, and that's a facility not generally available elsewhere. Releasing it under the GPL and limiting its use to free programs gives our community a real boost. At least one application program is free software today specifically because that was necessary for using Readline"[3]. Of course, the other possibility is that this strategy will backfire, and the code will not be used much.

LGPL is probably one of the most confusing open source licenses to read. While most companies find it easy to comply with, that is because those companies have adopted a simplified rule of compliance: only use LGPL code as a dynamically linked library. However, those who read the document usually ask, "Where does it refer to dynamic linking?" and in fact, it does not expressly do that.

LGPL is essentially GPL plus additional permissions that allow integration of LGPL libraries with proprietary applications. In version 3.0 of the licenses, LGPL was drafted as an addendum to GPL, which is

[3] From its website post "Why you shouldn't use the Lesser GPL for your next library," www.gnu.org/philosophy/why-not-lgpl.html

probably the right way to make it understandable. However, LGPL 2.1, the most common version in use, is a separate document, and it can be hard to track against GPL version 2.

In truth, the terms of the license are more complex. If you want to go beyond this bright-line rule for compliance, take a look at Chapter 9: LGPL 2.1 Compliance.

Corporate-Style (or "Weak") Copyleft Licenses

There is a category of weak copyleft licenses that are written to implement the copyleft principles of GPL, but with a drafting style more familiar to licensing attorneys. These licenses include the Eclipse Public License (EPL), the Mozilla Public License (MPL), and the Common Development and Distribution License (CDDL).

MPL covered the release of the Netscape web browser in 2002. This software was later improved and released as the Firefox browser, also released under MPL. This license was revised and updated in 2012 and released as MPL version 2.

The IBM Public License was drafted soon after MPL 1.0, but it has since been superseded by the CPL and EPL. The EPL mainly covers the Eclipse development environment.

The Sun Industry Standards Source License was released by Sun Microsystems, but it was superseded[4] by the Common Development and Distribution License (CDDL) in 2005.

These licenses roughly track the substance of LGPL. Although they use different exact wording to describe the scope of their copyleft obligations, none of them is considered as broad in scope as GPL. They are therefore accurately referred to as *weak copyleft* licenses—they allow code libraries released under their licenses to be incorporated into proprietary products. In addition, many of them allow relicensing of binaries under proprietary terms, as long as the source code remains

[4] http://en.wikipedia.org/wiki/Sun_Industry_Standards_Source_License.

available under the open source license. Therefore, in a sense, the copyleft only applies to source code. This approach is corporate friendly, obviating the need for the license carve-outs described above for LGPL.

All of these licenses also have express patent licenses and defensive termination provisions. (A summary of these terms is in Chapter 13: Open Source and Patents.)

Permissive Licenses

There are hundreds of small variations on permissive licenses, but almost all of them are based on the template forms of the BSD (Berkeley Software Distribution) license or the MIT license, or they consist of the terms of Apache 1.1 or 2.0. Most permissive licenses are short and simple. The simplest form of BSD appears below:

> * Copyright (c) <year>, <copyright holder>
>
> * All rights reserved.
>
> * Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:
>
> * * Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
>
> * * Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
>
> * * Neither the name of the University of California, Berkeley nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.
>
> * THIS SOFTWARE IS PROVIDED BY THE REGENTS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE REGENTS AND CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR

> CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Many lawyers are confused by the permission to use software in "source code and binary forms." They wonder whether this distribution must be in both forms or neither. (It would have been more accurate to allow redistribution and use in "source code or binary forms.") Some lawyers are also troubled by the lack of a formal license grant. In practice, neither of these issues is a serious concern; BSD is clearly a permissive license and intended to be an irrevocable grant of rights. This is an example of why open source licenses need to be interpreted in context rather than in isolation; any lawyer advising his clients to be concerned about these issues is not seeing the whole picture and not doing his client any favors.

Some readers of this license also wonder why there is an asterisk at the beginning of every line. In many programming languages, asterisks indicate comments—text that will be ignored by the compiler because it is not intended as programming statements. Mostly, this format is an artifact; licenses are usually in their own text files today, but this format allows one to put the license statement in a source code file.

The MIT license appears below:

> Copyright (c) <year> <copyright holders>
>
> Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:
>
> The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

> THE SOFTWARE IS PROVIDED "AS IS," WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NON-INFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

Apache

Apache 1.0 was deprecated because of its so-called "advertising requirement," which read as follows: "All advertising materials mentioning features or use of this software must display the following acknowledgment: This product includes software developed by the Apache Group for use in the Apache HTTP server project (http://www.apache.org/)."[5] This requirement was viewed as unworkable and possibly inconsistent with GPL. Very little software is still provided under this license. Apache 1.1, released in 2000, superseded Apache 1.0 and removed the advertising clause.[6] It is very similar to the BSD and MIT licenses.

Apache 2.0, released in 2004, was revised to standardize drafting and add patent provisions. Apache 2.0 has become the template license of choice for many open source projects, including the Google Android project.[7]

Miscellaneous Licenses

There are many other open source licenses, most of which are used on few projects. One category of license that's notable for its amusement value is sometimes called *otherware*. Licenses in this

[5] www.apache.org/licenses/LICENSE-1.0.
[6] www.apache.org/licenses/LICENSE-1.1.
[7] www.apache.org/licenses/LICENSE-2.0.

category are generally thought to be permissive, although serious discussions do sometimes crop up over whether they meet the open source definition, due to their tendency to impose odd conditions that may be considered restrictions on use (and therefore inconsistent with the open source definition). Some have described these licenses as social commentary on the length and complexity of the GPL. Some examples of *otherware* licenses are below:

The "Beer-Ware" License

> # 'THE BEER-WARE LICENSE' (Revision 42)
>
> # <**tobez@tobez.org**> wrote this file. As long as you retain this notice you
>
> # can do whatever you want with this stuff. If we meet some day, and you think
>
> # this stuff is worth it, you can buy me a beer in return. Anton Berezin[8]

The BarCamp License (Revision 1)

> <**tyler@bleepsoft.com**> wrote this code. As long as you retain this notice you can do whatever you want with this stuff. If we ever meet at a BarCamp, and you think this code is worth it, you can buy me some tacos in return. -R. Tyler Ballance[9]

The Catware License

> This program is catware. If you find it useful in any way, pay for this program by spending one hour petting one or several cats.[10]

The Chicken Dance License

This license was intended to "bring humor to the silliness of intellectual property" and is maintained at a GITHUB site: http://github.com/supertunaman/cdl. The license is based on BSD, with the following condition for redistribution:

[8] www.tobez.org/download/port-tools/port-idea .

[9] http://tyler.geekisp.com/code/BarCampLicense.txt.

[10] http://lists.debian.org/debian-devel/1999/01/msg01921.html. Note the query as to whether this qualifies as a "free" software license.

4. An entity wishing to redistribute in binary form or include this software in their product without redistribution of this software's source code with the product must also submit to these conditions where applicable:

* For every thousand (1000) units distributed, at least half of the employees or persons affiliated with the product must listen to the "Der Ententanz" (AKA "The Chicken Dance") as composed by Werner Thomas for no less than two (2) minutes

* For every twenty-thousand (20000) units distributed, one (1) or more persons affiliated with the entity must be recorded performing the full Chicken Dance, in an original video at the entity's own expense, and a video encoded in OGG Theora format or a format and codec specified by <OWNER>, at least three (3) minutes in length, must be submitted to <OWNER>, provided <OWNER>'s contact information. Any and all copyrights to this video must be transfer[r]ed to <ORGANIZATION>. The dance featured in the video must be based upon the instructions on how to perform the Chicken Dance that you should have received with this software.

* Any employee or person affiliated with the product must be prohibited from saying the word "gazorninplat" in public at all times, as long as distribution of the product continues.

A license that has been trending upward in popularity in the past few years is the WTFPL:

Do What The Fuck you want to Public License

Version 1.0, March 2000

Copyright (C) 2000 Banlu Kemiyatorn (]d).

136 Nives 7 Jangwattana 14 Laksi Bangkok

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

Ok, the purpose of this license is simple and you just DO WHAT THE FUCK YOU WANT TO.

Unfortunately, the terms don't specify what you can do with the code. But one presumes the author does not care enough to challenge you. Also, the FSF considers this license compatible with GPL.[11]

There are also several kinds of public domain dedication, such as the Unlicense and Creative Commons Zero. These are not licenses, of course; they are statements of intent to waive license conditions along with the copyright for the code.

OpenSSL

This chapter will not attempt to address the many unusual or nonstandard permissive licenses. However, one nonstandard license, the OpenSSL license, is extremely common because it applies to a project in ubiquitous use. OpenSSL is an open source implementation of a secure socket layer toolkit. The OpenSSL license looks like a BSD-style permissive license at first blush, but it contains language that many readers find confusing. The project website says:

> The OpenSSL toolkit stays under a dual license, i.e., both the conditions of the OpenSSL License and the original SSLeay license apply to the toolkit.[12]

This is the less common use of *dual license*—that two licenses simultaneously govern the code. Previous FAQs stated that OpenSSL was a "BSD-style" license, and that is how it is generally treated in the industry. However, OpenSSL's new licensing FAQ[13] contains suggestions for resolving the conflict between the OpenSSL and GPL licenses: "If you develop open source software that uses OpenSSL, you may find it useful to choose another license than the GPL, or state explicitly that 'This program is released under the GPL with the additional exemption that compiling, linking, and/or using OpenSSL is allowed.'" This suggests there is a conflict between the two to resolve, which would not be the case if OpenSSL were a true BSD-style license.

[11] See the list of compatible licenses at www.gnu.org/licenses/license-list.html#WTFPL.

[12] www.openssl.org/source/license.html.

[13] www.openssl.org/docs/faq.html.

The FAQ goes on to imply that it is the GPL authors, not the OpenSSL authors, who believe there is a conflict: "Some GPL software copyright holders claim that you infringe on their rights if you use OpenSSL with their software on operating systems that don't normally include OpenSSL."

> The license and distribution terms for any publicly available version or derivative of this code cannot be changed; i.e., this code cannot simply be copied and put under another distribution license (including the GNU [General] Public License).[14]

This confusion illustrates the drawbacks of nonstandard licenses—many readers are concerned that the language above means the license is intended to be a copyleft license.

This license has, fortunately, now been deprecated by its own project. However, it persists because there are legacy versions of OpenSSL in use.

Content Licenses

Software packages contain plenty of non-software materials, such as bitmapped images (like icons), music files, picture files (GIFs, JPEGs, etc.), and text files—which are sometimes referred to as *content*. When the software is open source, it is helpful to have an equivalent non-software license for these materials. Also, authors who want to make content freely available, independent of software, need licenses to effectuate that release. The most common of these are the GNU Free Documentation License[15] and the Creative Commons licenses.[16]

The GNU Free Documentation License is FSF's equivalent of GPL for a "manual, textbook, or other functional and useful document" or "any textual work, regardless of subject matter or whether it is published as a printed book ...principally for works whose purpose is

[14] Spelling and punctuation made to conform to US usage.
[15] www.gnu.org/licenses/fdl.html.
[16] http://creativecommons.org/.

instruction or reference."[17] Many of this license's terms are specific to documentation and do not apply well to other works. It is not in common use.

Using the Creative Commons licenses is a more popular choice for any kind of non-software copyrightable work. These licenses offer variations to suit the breadth of rights the author wishes to grant; accordingly, not all of them are consistent with the Open Source definition. For instance, there is a "noncommercial" (NC) option that conflicts with the rule that open source licenses must not place any restrictions on fields of endeavor.[18] The licenses include options that disallow modification (*no-derivatives*). All of them require attribution (referred to as *by*), and some contain copyleft conditions (called *ShareAlike*). Each license comes in two versions—a summary and the *legal code*, which contains the actual license terms. Great care has been taken to make the application of these licenses as international as possible.[19]

The Creative Commons Public Domain Dedication (CC0) is particularly popular for software because it disclaims any waiver of patent rights. This is a handy way for companies to make available copyrightable software for broad use without undue concern about whether any patent rights have been waived. Or, taking the other view, it's a handy way to trick the world into using patented code. But mostly, that is not the intention; it is more to avoid tangential arguments that the company's patents have been licensed free of charge and therefore cannot be enforced at all. (For more discussion on this point, see Chapter 16: Open Source Releases.)

Other than the permissive (BY) and public domain (CC0) versions, Creative Commons licenses are generally not a good choice for release of software. The scope of the ShareAlike versions does little to enlighten a licensee about what other code might be controlled as an

[17] Preamble to GFDL.

[18] "6. No Discrimination Against Fields of Endeavor," http://opensource.org/osd-annotated/

[19] The revision 4 licenses in particular contained internationalization terms; see http://creativecommons.org/version4/

"Adaptation."[20] In other words, as confusing as the scope of GPL is, CC ShareAlike is even more confusing when applied to software.

Problematic Licenses

A few licenses are worth noting because they almost always cause compliance concerns. I would put these in the category of my least favorite licenses.

- **ODbL.** An "open data" license with copyleft conditions. (See Chapter 21: Open Hardware and Data.)
- **CPAL.** A "badgeware" license that was approved by OSI while the "Exhibit B" licenses (variations of MPL with badgeware requirements) pioneered by Sugar CRM (and later deprecated by it) became popular. Badgeware licenses cause most companies compliance concerns because their conditions can apply in the absence of distribution. In this respect, they can be considered ultra-strong copyleft licenses similar to APGL.
- **A "public domain" license.** This is on its face a dedication to the public domain, which requires a copyright or licensing notice. This "license" is self-contradictory because no copyright notice is appropriate on a work to which the author has waived copyright protection, and fortunately, it's not very common.

[20] For the definition of Adaptation in a ShareAlike Creative Commons license, see, for example, Section 1a, http://creativecommons.org/licenses/by-sa/4.0/legalcode.

Chapter 4

License Compatibility

The need for due diligence and the question of license compatibility go far beyond open source licensing. Lawyers have been addressing issues of software license compatibility since software licensing began; the issues have only been brought to the forefront by open source licensing, which has caused compatibility to be a concern for everyone—not just the lawyers. Much of what you will read in this chapter is about software licensing in general, and in the end, you may come to the conclusion that it is proprietary licenses, and not open source licenses, that cause the most problems in due diligence. Today, open source licensing results in extensive due diligence, mostly because there is so much software under open source licenses.

The Awkward Dinner Party

Developing with different kinds of software is like planning an awkward dinner party for your relatives. By duplicating a lot of effort, you can feed everyone: your middle-aged uncle, on his low-carb diet, wants meat and fish; your sister, the vegan, wants only locally grown vegetables; and your teenaged nephew will eat anything as long as it is pizza. But what if your dinner guests not only limited their own diets but also harbored a vehement, polemical disgust for everything they wouldn't eat? It would be hard to bring everyone to the same table.

These woes, like many that populate our world's headlines, come not from the participants being different but from the participants refusing to coexist with others. Software licenses, like people who are certain that they are right and everyone else is wrong, each have their

own set of rules. When all the rules conflict, coexistence can be impossible.

What Is Due Diligence?

Most people who want to learn about open source licensing have a goal of conducting due diligence. This process has many names—audits, due diligence, housekeeping, compliance, and intellectual property hygiene. But whatever it is called, it is the process of ensuring—as much as possible—that your company is complying with the open source licenses that cover the software it is using. In this chapter, I will refer to this process as *diligence*.

A diligence project can arise for many reasons. It almost always comes up during a corporate transaction—a merger, acquisition, divestiture, or financing deal. But it can also arise because of customer requirements, regulatory audits, or simply an initiative to comply with licenses in order to manage risk or even to do the right thing by respecting the intellectual property of others.

The diligence process is not about perfection; it is about risk management. There is no such thing as perfect compliance; the software landscape, even for a simple product or business, is too complicated for that. The process of diligence is designed to solve the worst problems first, then move to the next set, and then move to the next until one runs out of time, energy, or fear of risk. It is a process of triaging problems and making reasoned decisions.

From an overarching point of view, diligence is the process of making sure your inbound rights are equal to or greater than your outbound rights. By inbound, we mean licenses granted to your company, and by outbound, we mean rights exercised by your company or granted to others. If you grant, or use, more rights than you have, then you are infringing on someone else's rights.

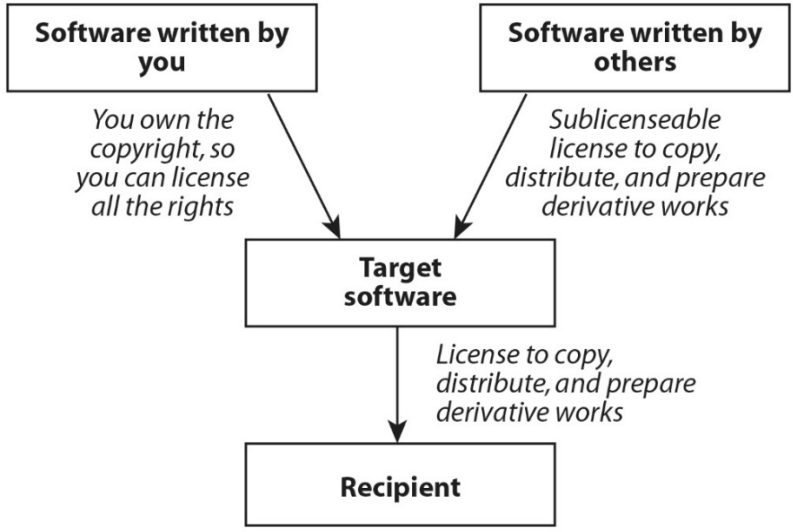


Figure 4.1 Two common cases governing clearance of inbound rights

Figure 4.1 shows two common cases governing the clearance of rights going into a software code base (i.e., the inbound rights). One is software written by the party creating the target software. As an author, that party has the right to exercise the copyright. The other element is software that is written by others—and that therefore requires a license to exercise the copyright—but that has been provided under a broad inbound license. Each of these inbound cases is sufficient to clear the rights to license out to the recipient—meaning the outbound rights. As long as the outbound rights are less than or equal to the inbound rights, the licensing works.

Potential Diligence Issues

Problems with clearance of rights can come in many forms, but the most common are license restrictions and license conditions. License restrictions only occur in proprietary licensing; license conditions are more commonly an issue in open source software licensing.

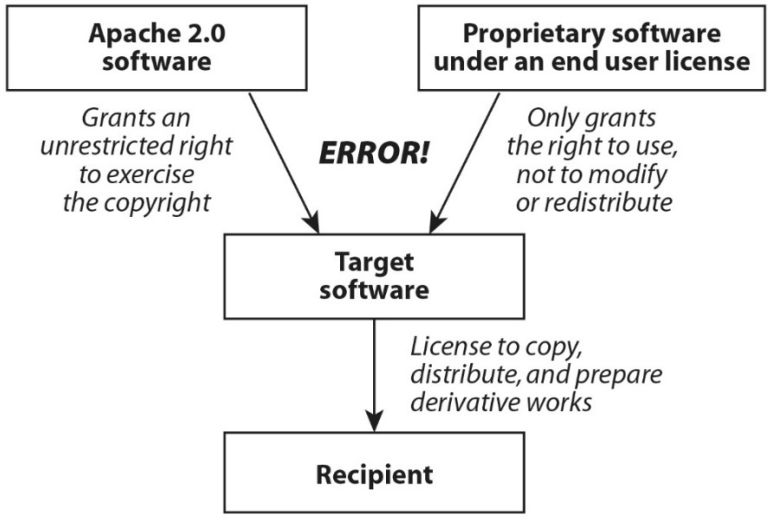


Figure 4.2 License restriction problems

Figure 4.2 depicts the quintessential diligence problem in proprietary licensing. The scope of the inbound license for the proprietary software is narrower than that of the outbound grant; some of the rights granted out are not granted coming in. Of course, this problem can't happen for open source software components because the open source definition requires that there be no restrictions on the license grant—that's Freedom Zero. However, open source licenses can impose conditions that create diligence problems, as Figure 4.3 shows.

In Figure 4.3, the software code base includes a component whose inbound license is covered by GPL. Because GPL is a copyleft license, the copyleft license conditions must be followed when redistributing the software. But here, there has been a mistake; those conditions have not been flowed down to the recipient. This is the quintessential open source diligence problem. Another way to say this is that GPL as an inbound license is not compatible with Apache as an outbound license.

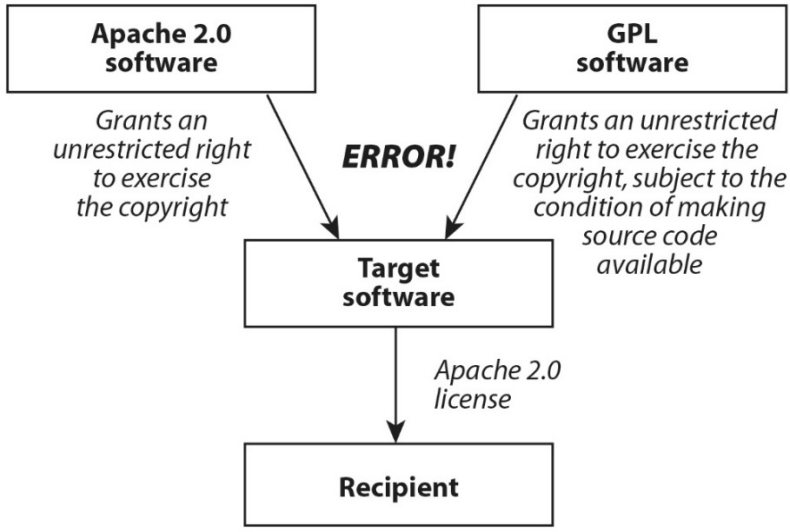


Figure 4.3 License condition problems

In open source software, conflicts of conditions create many diligence problems. A company that has developed the target code in the figure above would have to change its outbound license to GPL to solve the problem. To create software whose licensing works correctly, we need to use only inbound licenses that are compatible with the outbound license. Accordingly, only inbound licenses with fewer and consistent conditions should be used, as compared with the outbound license. When those in the open source world talk about compatibility, this is what they mean.

In the example in Figure 4.4, many licenses that are compatible with GPL 3 can govern this project. The basic rule is that the outbound license must be the one with the most conditions.

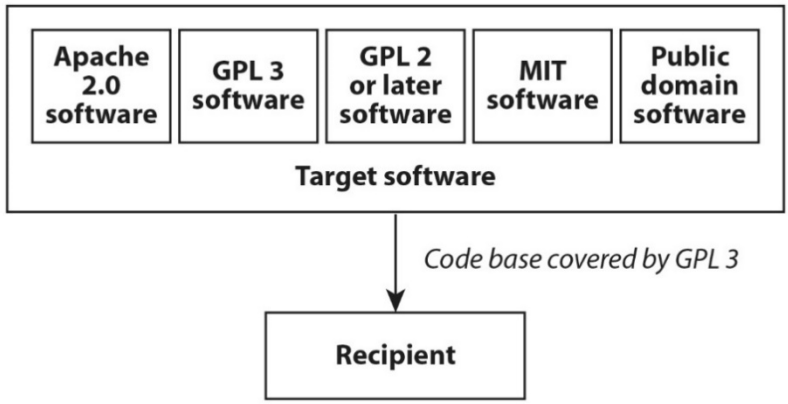


Figure 4.4 License compatibility

However, components governed by licenses such as AGPL (which has more conditions), Eclipse Public License, or CDDL (which all have different copyleft conditions) could not be included in this software. All of these copyleft licenses are like the guests at the awkward dinner party—their diets are mutually exclusive. This is because all of them exclude placing additional licensing restrictions on the software. Also, because each of them contains slightly different terms, each consists of additional restrictions vis-à-vis the others. For example, Section 6 of GPL 2 says:

> Each time you redistribute the Program (or any work based on the Program), the recipient automatically receives a license from the original licensor to copy, distribute or modify the Program subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein.

A few copyleft licenses are compatible. For example, LGPL code can be redistributed under the corresponding version of GPL because LGPL is merely a set of additional permissions that allow certain kinds of integration with non-GPL software. In the version 3 licenses, this is more obvious, as LGPL 3 is drafted as a set of additional terms to GPL

3; for the version 2 counterparts, Section 3 of LGPL 2.1 contains a specific provision allowing this:

> You may opt to apply the terms of the ordinary GNU General Public License instead of this License to a given copy of the Library. ... Once this change is made in a given copy, it is irreversible for that copy, so the ordinary GNU General Public License applies to all subsequent copies and derivative works made from that copy. This option is useful when you wish to copy part of the code of the Library into a program that is not a library.[1]

Some copyleft licenses, like MPL, contain specific provisions that make them compatible with other copyleft licenses. For example, MPL 2.0 says:

> 3.3. Distribution of a Larger Work
>
> You may create and distribute a Larger Work under terms of Your choice, provided that You also comply with the requirements of this License for the Covered Software. If the Larger Work is a combination of Covered Software with a work governed by one or more Secondary Licenses, and the Covered Software is not Incompatible With Secondary Licenses, this License permits You to additionally distribute such Covered Software under the terms of such Secondary License(s), so that the recipient of the Larger Work may, at their option, further distribute the Covered Software under the terms of either this License or such Secondary License(s).

The Secondary Licenses include GPL and LGPL, and this provision creates compatibility by fiat with them, unless the author specifically elects to eschew compatibility by applying a notice stating that the software is not available under the Secondary Licenses. Given that most adopters rarely elect to limit applicability of Secondary Licenses, or later versions, this clause created broad compatibility among the most common versions of MPL and GPL.

The above discussion has to do with what I think of as vertical compatibility—given an inbound license, can the software covered by it be redistributed in a code base covered overall by another outbound license? But keep in mind that open source licensing is not a

[1] http://opensource.org/licenses/lgpl-license.php.

sublicensing regime. The inbound license terms do not change; they are actually passed through directly to all recipients. However, in open source, all of the rights of copyright are granted, so there cannot be any discrepancy between the licenses granted; the only difference is the conditions imposed on exercise of the license. Therefore, as long as the conditions of the inbound license and the outbound license are not mutually exclusive, there is no compatibility problem.

"Horizontal" Compatibility Issues

However, there is a more subtle issue that I think of as horizontal compatibility, and that issue only arises under GPL (or AGPL) and LGPL, because those are the only licenses that place limitations on how code can be integrated. There will be more details in Chapters 8 and 9 on GPL and LGPL compliance, but for now, the shorthand rules are as follows:

- If any code in a program is GPL, it must all be provided under GPL.
- LGPL code should be integrated into a program with other code only as a dynamically linked library.

Figure 4.5 shows the problem that can arise from a horizontal incompatibility. In this case, the program contains code covered by inbound terms under various licenses. But the code under the copyleft licenses cannot be redistributed under any other terms, so there is no one license that will work.[2] This is like the dinner table where no one meal will satisfy anyone. The GPL is the diner who not only will not eat the same meal as the others but also will not tolerate the presence of other meals at the same table.[3]

[2] For discussion of the actual or theoretical incompatibility between Apache 2.0 and GPL 2.0, see below in this chapter.

[3] The same is true for many proprietary licenses as well, of course.

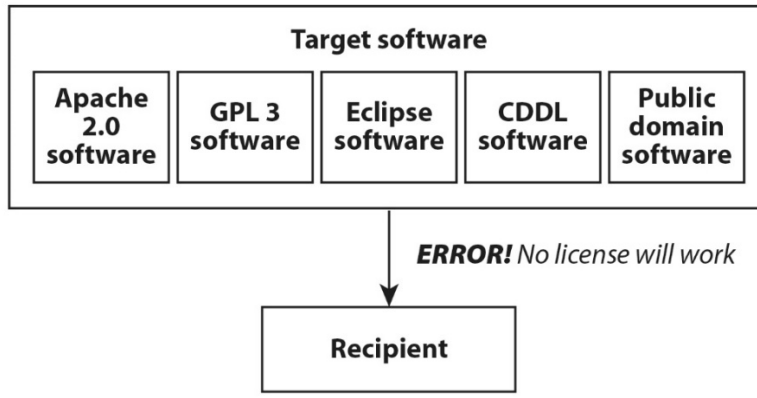


Figure 4.5 Problem due to horizontal incompatibility

In contrast, software provided under weak copyleft licenses can often coexist in the same program, and of course, permissive licenses place no restrictions on other code. So, the situation shown in Figure 4.6 works. Because all the licenses are copyleft, but weak copyleft, the licensing to the recipient can be achieved by passing through the licensing terms simultaneously. Each component will be governed by its own license, and the code base as a whole has no one license. This is like the dinner table where each visitor eats his own meal, and while the visitors do not share, they can sit at the same table.

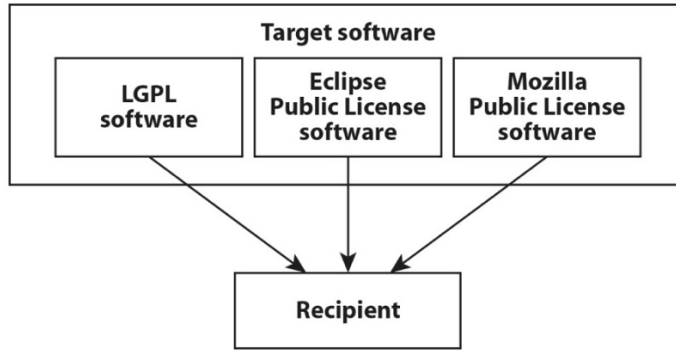


Figure 4.6 Horizontal compatibility

How to Avoid License Bugs

All of the above scenarios view diligence from the point of view of a developer that is trying to create a software product using components of open source software that are licensed by third parties. In such a case, the developer mostly needs to take the inbound licensing as a given. However, some developers will have the option of releasing their own software under a license not dictated by inbound licensing concerns. (For more discussion of this option, see Chapter 16: Open Source Releases.) The above discussion shows one aspect of why this option is so important. A developer company can license its software under whatever terms it likes, but if it makes certain choices, like choosing GPL for a library that must be included in the same program with proprietary or other copyleft components, it may create a situation where other developers cannot reuse the software in other projects. Sometimes, taking this option is a way to force users to buy proprietary licenses, as in a dual licensing model. In such cases, the developer is purposely creating a license bug and a path to resolve it. But if such a choice is made without clear thought, the licensing could backfire and merely discourage adoption of the software.

Apache 2 and GPL 2

It was noted above that Apache 2 and GPL 2 may be incompatible. On its face, this may seem hard to explain. Permissive licenses are generally compatible with any other open source license. However, after Apache 2 was released, the FSF took the position that Apache 2.0 was incompatible with GPL 2. Information about this difference of opinion—between open source organizations with very different philosophies—is a bit hard to come by. But Apache Foundation posted the following about it:

> After spending a couple hours on the phone with the FSF, we have a better understanding of the particular interpretation of the GPL that might lead one to construe the following: granting an explicit patent license causes any implicit patent licenses to be null and void; revoking that explicit patent license causes the person who is claiming

> infringement of their patent to lose the patent rights that would otherwise have been attained via the GPL's implicit rights; loss of patent rights means loss of right to use; GPL section 7 allows a patent owner to claim infringement of a patent within a GPL'd work and continue to distribute that work as GPL up until a third party imposes a restriction on the rights of others to distribute (i.e., until a judgment or injunction is placed on the work).
>
> GPL section 6 saying "You may not impose any further restrictions on the recipients' exercise of the rights granted herein" does not apply to patents because the "rights granted herein" are only copyright.[4]
>
> This is our current understanding of the position held by the FSF; whether or not our understanding is correct has not yet been confirmed.
>
> Note that this is contrary to our previously stated belief that the GPL does forbid the continuing use of a GPL'd work by an entity that has claimed the work contains infringement of their own patented technology. Apparently, it is okay for the distribution and use to continue up until a judgment or injunction has been issued because the FSF does not believe a claim of patent infringement amounts to a restriction on the rights of others to redistribute, and the constraint on further restriction applies only to those rights listed within the GPL itself (copyright).

Apache has stated that it "considers this issue to be in legal limbo, at least until we get a definitive answer regarding the survivability of implied patent licenses."[5]

The FSF's position on express patent licenses' taking away implicit licenses—as least as characterized above—is not a foregone conclusion under law; the position is probably advocacy as opposed to objective legal analysis. The issue of patent license termination is more interesting, in light of the so-called "liberty or death" provision in Section 7 of the GPL:

> If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute

[4] The same is true for many proprietary licenses as well, of course.

[5] http://people.apache.org/~slive/site/xdocs/licenses/GPL-compatibility.xml.

> so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Program at all.

The Apache Foundation points out[6] that even if we accept the FSF's reasoning, an additional restriction will only arise if there actually are patents covering the work that would otherwise be licensed. The Apache Foundation, which is the licensor of a great deal of code under the Apache 2 license, owns no patents,[7] and it is likely that many licensors who release code under Apache 2 are in the same position; most companies that release code under Apache 2 do not choose to do so for code covered by patents the company values.

Note that GPL 3 is understood by FSF to be compatible with Apache 2.0.[8]

This issue (actual or effective) resulted in an interesting license exception for the LLVM project, which develops a widely used development tool.

> [I]f you combine or link compiled forms of this Software with software that is licensed under the GPLv2 ("Combined Software") and if a court of competent jurisdiction determines that the patent provision (Section 3), the indemnity provision (Section 9) or other Section of the License conflicts with the conditions of the GPLv2, you may retroactively and prospectively choose to deem waived or otherwise exclude such Section(s) of the License, but only in their entirety and only with respect to the Combined Software.

This essentially says that if a court ever agrees with FSF that the licenses are incompatible, the licensee need not absorb the risk that it has violated GPL by combining code under the two licenses. LLVM took this approach because it needed to give users the comfort of Apache 2.0's patent license, but allow them to build files using GPL code.[9]

[6] http://people.apache.org/~slive/site/xdocs/licenses/GPL-compatibility.xml.
[7] http://people.apache.org/~slive/site/xdocs/licenses/GPL-compatibility.xml.
[8] http://gplv3.fsf.org/rationale/.
[9] For more explanation, see https://llvm.org/foundation/relicensing/.

License Proliferation

No discussion of license diligence would be complete without a discussion of *license proliferation*. It is a divisive subject: standardization of license terms unarguably makes the diligence process easier; however, those arguing against proliferation often commit the logical fallacy of equating choice with fiat, assuming that uniformity of licensing terms should turn on their own choice. Those who consider their licenses the best choice for everyone have the right to their opinion, of course, but they do not have the right to choose for other authors, who have the right to choose their own terms.

Anyone who complains about proliferation in open source licensing has probably not spent much time doing diligence on proprietary licenses. Proprietary licenses have a dizzying array of license restrictions—as opposed to licensing conditions—that make diligence expensive and time-consuming. An entire industry of license management and software deployment tools exists to help enterprises comply with proprietary license restrictions. Open source may have over 60 licenses, but proprietary licenses exist in an infinite variety.

Others who take a more nuanced view are not so much against new open source licenses as against new licenses that are difficult to understand or poorly written. Sadly, many open source licenses are so badly drafted that they are maddeningly difficult to understand, and although proprietary licenses can also be badly written, they tend to get updated and improved over time as they expire and are renegotiated. But open source licenses are forever.

Regardless of your opinion on license proliferation, if you are releasing open source software, you should resist the urge to write a new open source license. Doing so is very unpopular and, especially for copyleft licenses, very challenging. Most companies releasing open source code find that one of the existing licenses works for them (at least moderately well) and that the faults of the existing licenses are outweighed by the goodwill engendered by easing the task of review for those recipients expected to embrace the software.

Chapter 5

Conditional Licensing

Open source licenses are conditional licenses. This concept is one of the most difficult for open source licensing novices to grasp, and an insistence on the importance of grasping it is not just pedantry. The opacity of this concept underlines a serious misconception about open source licensing. Like the shadows of Plato's cave, misconceptions about conditional licenses cause far more fear than actual danger. Some people refer to the GPL as "viral," and use of this word is not mere political incorrectness or rhetoric; it exaggerates the risks attendant on using open source software. Eradicating the word *viral* from discussions of open source licensing is long overdue because this word has skewed and limited understanding of open source licensing principles.[1]

It's Not a Virus, It's a Bug

The single most frequent question I hear from clients about GPL is: "How can we keep it from contaminating our proprietary code?" The answer is easy: it can't contaminate your proprietary code. There is simply no plausible legal argument as to how it could do that. But in this chapter, we will examine the question in detail, why the misconception persists, and why the conditional licensing model limits the worst-case scenario for GPL violations.

As we know from the prior chapter, it is possible to violate GPL by integrating GPL code with proprietary code and distributing that combined code under terms other than GPL. Developers, with visions

[1] I have heard several stories about how this word began to be associated with free software in general or the GPL in particular, but I can't verify them, so I won't repeat them here.

of viruses dancing in their heads, worry that if GPL and proprietary code are combined into a single program, the GPL licensing terms will "infect" the proprietary code and the proprietary code will be automatically licensed under GPL. The developers then worry that they will be obligated to provide the source code to their own proprietary code.

For most proprietary code developers, this is simply not an option. Not only would doing this violate their fiduciary duty to their shareholders by devaluing their proprietary code, but it would probably make them violate third-party licenses. A company's proprietary product usually contains third-party software that is sublicensed under proprietary terms. The company therefore has no right to convert that product to GPL, even if it wanted to.

But "infection" is not how copyleft works. In fact, if a developer were to combine GPL and proprietary code in a way that violated GPL, then the result would be that GPL had been violated—no more, no less. (And if the developer combined third-party GPL code with third-party proprietary code, then both inbound licenses would have probably been violated.) What that means, legally, is that the author of the GPL code may have remedies for violation of GPL and, if the license is terminated as a result, remedies for unlicensed use of the GPL software. Both possibilities, essentially, are copyright infringement claims. The legal remedies for a copyright infringement claim are damages (money) and injunction (stop using the GPL code).

In fact, there is simply no legal mechanism for GPL changing the licensing terms for other code. For software to be licensed under a particular set of terms, the author has to take some action that reasonably leads a licensee to conclude that the licensor has chosen to offer the code under those terms. In contrast, combining proprietary and GPL code in a single program in violation of GPL is a license incompatibility—meaning the two sets of terms conflict and cannot be satisfied simultaneously. The better analogy for this kind of licensing incompatibility is a software bug, rather than a software virus.

What Is a Conditional License?

The GPL and almost all other open source licenses grant a copyright license simultaneously to all who wish to take the code under that license. But that license is conditioned on complying with some requirements. Those requirements range from a license notice (for permissive licenses) to copyleft conditions (making source code available, for copyleft licenses). If those conditions are violated, the license terminates.

One reason for the struggle with this idea is that conditional licenses are not very common other than in the landscape of open source licensing. The closest familiar analog for lawyers is probably a unilateral contract. Every law school contracts class starts with this hypothetical: "If you cross the bridge, I will give you $10." When you cross the bridge, the $10 is due. But open source licensing is the opposite. Instead of saying, "Perform this task and I will reward you," it says, "Enjoy this boon until you misbehave." Legally, these are different.

License or Contract?

You may hear open source advocates say that an open source license is "not a contract." A better analysis is that open source licenses are not necessarily contracts, but this is not the same as saying they are never contracts. To pose the question, "License or contract?" oversimplifies the issue. But first, let's understand the distinction between a contract and a conditional license.

A *contract* is a set of promises. Under the law, a contract is formed when there is an offer, an acceptance, and consideration. Consideration is something of value exchanged for a promise. The law will not recognize and enforce a contract unless each party has promised something. (Otherwise, the arrangement is called a *gift*.) The law does not require the bargain to be fair or even,[2] only to be what the parties

[2] In law, consideration is sometimes referred to as a "peppercorn"—in other words, it doesn't matter whether the thing given is of value, so a peppercorn will be enough. Like many legal

intended. One form of consideration is forbearance—not doing something the party would otherwise have the right to do.[3]

License contracts therefore always involve forbearance because a license is a promise not to sue for intellectual property infringement. The licensor promises not to sue, and the licensee promises something in return, such as the payment of a license fee. Most intellectual property lawyers are accustomed to this model. But open source licenses are subtly different.

In an open source license, there is no consideration per se given by the licensee. Instead, there are conditions for exercising the license. So long as the licensee abides by these conditions, the licensee can continue to exercise the license. If the licensee breaches the conditions, the license disappears, and the licensor is free to sue for infringement.

The difference between a conditional license and a contract is subtle but important. In a contract, each party makes a promise, and if a party breaks the promise, a court can order that party to fulfill it. In truth, courts almost never order people to perform contracts. If someone breaches a contract, the court orders the breaching party to pay damages. In fact, the law almost never supports anything else. Ordering a party to actually perform a contract is called *specific performance*. This is an extraordinary remedy that is only supported in limited circumstances, such as sales of real property. For the most part, breach of contract is considered a purely commercial harm, and accordingly, the only remedy is money. This notion is important, and it actually speaks to a basic premise of political freedom in a common law country. For a civil wrong (as opposed to a criminal one), the punishment is payment of money or an order to stop doing something. It is an important tenet of our political freedom that courts cannot order us to take positive action except in extraordinary cases.[4]

terms, *peppercorn* has long outlived its original sense. At the time the term was coined, pepper was an expensive, imported luxury. See Restatement (Second) Contracts Section 79.

[3] Such as paying someone for not smoking. See Restatement (Second) Contracts Section 71.

[4] In criminal law, courts can imprison or take other actions to curtail personal liberty, but that is different because criminal law seeks to redress harm to society rather than to individuals. It therefore has remedies that are not available in civil law.

In a conditional license, the licensee makes no promises to do anything. In fact, an open source license grants you rights whether you intend to exercise them or not. You, and I, and everyone else are licensed under GPL the moment the licensor releases the code under that license. However, actually exercising that license has conditions: the licensee must abide by the conditions of the license or the license will be lost. As GPL 2 Section 5 says:

> You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Program or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Program (or any work based on the Program), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Program or works based on it.

But in truth, this is not an attempt to form a contract so much as to explain that the licensee must either abide by the terms or forego the benefits of the license.

If this view seems tortured, perhaps it is. The "license not contract" position was pioneered alongside GPL 2 during the 1990s. In the early years of that decade, the law about online contract formation was still unsettled. Traditionally, entering into contracts involved a written signature—this was the means of expressing acceptance of the contract's terms. But as the software industry grew, software developers sought to establish the terms of license agreements for a mass market without the need for paper documents and signatures. The use of "shrink-wrap" or click-to-accept licenses was in common practice by the 1990s, but the efficacy of these methods to form a contract had still not been confirmed by the courts. Then in 1996, a landmark opinion was issued upholding the enforceability of these unsigned agreements.[5] But forming a contract in this way still required the licensor to leap some hurdles, such as implementing an installer or downloader that forced an "I accept" action from the licensee. Open source software is usually passed from one recipient to the next without any such technical

[5] *ProCD v. Zeidenberg*, 86 F.3d 1447 (7th Cir. 1996).

mechanisms, so the license terms need to have legal traction without the need to form a contract. The conditional license model worked well for this because it drew its power from copyright law, which gave the licensor the power to exclude certain uses of the software regardless of where, when, or how the licensee used it and regardless of the identity of the licensee and the means by which the licensee received the software.

Implications of the Conditional Licensing Model

The conditional licensing model has two main implications. First, this model offers different remedies from those available under a contract model. Second, the conditional licensing model does not create a contractual relationship (called *privity of contract*) between redistributors and their recipients and therefore places the rights and duties of enforcement on the original author of the software.

Remedies

The remedies for copyright infringement and breach of contract are quite different. In the United States, the damages for copyright infringement are "the actual damages suffered ... as a result of the infringement, and any profits of the infringer. In establishing the infringer's profits, the copyright owner is required to present proof only of the infringer's gross revenue, and the infringer is required to prove his or her deductible expenses and the elements of profit attributable to factors other than the copyrighted work."[6] The copyright plaintiff can also choose, as an alternative, "statutory damages" of $750 to $30,000 per copyrightable work, "as the court considers just." These damages can be enhanced to $150,000 if the plaintiff proves the infringement is willfully committed. Statutory damages are only available if the work has been registered with the copyright office on a timely basis.

[6] 17 USC 504.

Contract law, in contrast, allows the aggrieved party to seek compensatory damages only.[7] These damages are focused on economic harm and do not include compensation for emotional damages and pain and suffering (which are tort law concepts). If you breach a contract, you might pay expectation damages (profit the other party expected to receive from the contract), consequential damages (lost business or reputation caused by the breach), or restitution (an equitable calculation to avoid unjust enrichment of the breaching party).

Copyright also offers injunction as a remedy. An injunction is an order by the court to cease the infringement. When an injunction is available, as under copyright law, courts use a test with four factors: (1) the likelihood of the plaintiff's success on the merits; (2) irreparable harm in the absence of an injunction (i.e., the harm cannot adequately be compensated later by money damages); (3) the balance of hardships between the plaintiff and the defendant; and (4) the public interest.[8] In a contract claim, an injunction is an extraordinary remedy that is seldom available.

As you can see, the money damages for copyright violation are likely to be at least as high as those for breach of contract, and copyright adds the possibility of statutory damages and injunction. This is key in open source licensing, where the economic harm may be difficult to prove. Open source licensors give their software away royalty-free, so the harm to the licensor accrues from violation of the conditions, which are primarily notices and the requirement to make source code available. It may be hard to place an economic value on the harm to the licensor for failure to meet these conditions, and copyright law allows the licensor to elect other remedies. Therefore, foregoing the possibility of contract damages is no real loss to the licensor.

The virus model rests on the assumption that specific performance will be available as a contract remedy. For this to happen for an open source license, a contract would need to have been formed, the conditions of source code delivery would need to be interpreted as

[7] Punitive damages are extremely rare in contract law. Restatement (Second) of Contracts, §355.
[8] See, e.g., *Metro-Goldwyn Mayer, Inc. v. 007 Safety Products, Inc.*, 183 F.3d 10, 15 n.2 (1st Cir. 1999).

contractual obligations of the licensee, and most importantly, a court would need to order specific performance of those obligations. But as you can see from the above analysis, this should never happen. No licensee under a copyleft license would be ordered to lay open source code. This kind of specific performance is just not supported by the law. And copyright offers no such remedy at all. So there is no likely basis for such a remedy for breach of an open source license.

Privity

The second corollary to the conditional licensing model is that the licensor—or copyright owner—will always be the one to enforce the license conditions. This leads to a somewhat counterintuitive paradigm. If someone takes GPL code, modifies it, and refuses to provide the modified source code to a binary recipient, the recipient has no legal claim. Instead, it is the licensor of the GPL code who has the right to pursue a claim.

In the United States, only copyright owners or exclusive licensees have standing to sue for copyright infringement.[9] The conditional licensing paradigm enables the licensor to pursue enforcement without identifying licensees in advance. If open source licenses were enforced as contracts, then the copyright holder might need cooperation from the distribution channel to pursue enforcement. Any contract would have been formed between the recipient and the distributor, not the original licensor. Redistributors may have little incentive to enforce license conditions or may selectively do so for commercial gain, so the conditional licensing model also serves to empower the open source author more than the contract model might do.

The analysis above is enough to help you understand the virus fallacy and what you need to know about conditional licensing. The

[9] 17 USC Section 501(b) says, "The legal or beneficial owner of an exclusive right under a copyright is entitled ... to institute an action for any infringement of that particular right committed while he or she is the owner of it." Courts have held this statement to be exclusive under the doctrine of *expressio unius est exclusio alterius*. See, for example, *Silvers v. Sony Pictures Entertainment, Inc.*, 402 F.3d 881 (9th Cir. 2005) (en banc).

remainder of this chapter contains some more detailed legal analysis for those interested in the finer points of law surrounding this issue.

Contract Formation

As outlined above, most GPL licensors would prefer to cast violations as copyright infringements rather than breaches of contract. But those who say that open source licenses are categorically "not contracts" are probably wrong. Quite the opposite—in many cases, it might be easy to prove that a contract has been formed, but between whom and how might be complicated to prove.

If one sought to establish that a contract had been formed, then there is plenty to support that view. The express terms of most open source licenses suggest that the document is a contract. Most of them contain UCC warranty disclaimers,[10] which only apply to sales-of-goods contracts governed by Article 2 of the UCC. GPL 2 Section 5's statement "You indicate your acceptance of this License," if true, forms a contract, assuming that there has already been an offer and consideration. But most important, the circumstances surrounding acceptance of the terms may often be sufficient to support contract formation.[11] It's obvious that there has been an offer if the software is available for download. The consideration is also sufficient—the licensor has offered to forego infringement suits, and the licensee must comply with the conditions. The only challenging plank of formation is acceptance, and the UCC provides that acceptance can be indicated by "including conduct by both parties which recognizes the existence of such a contract." Such conduct is also likely to be easy to prove.

Incentives for Formation Arguments

The obvious question, then, is that if it is so easy to show acceptance via conduct, why was there so much uncertainty leading up to the ProCD case? It's important to keep in mind that the scope of

[10] For instance, GPL 2 Section 11.

[11] See UCC Article 2, Section 1-204.

open source licenses is quite different from that of EULAs. Accordingly, recipients of proprietary and open source software have very different incentives to avoid license terms.

A person who lawfully possesses a copy of software has the right to create backup copies (under 17 USC Section 1117(a)(2)) and transitory copies as necessary to run the software (under 17 USC 1117(a)(1)). In addition, under background law, someone who purchases a copy of software can invoke the first sale doctrine (17 USC Section 109), create copies as necessary to reverse engineer the software (in some limited circumstances where that use is a fair use),[12] and enjoy certain implied warranties under UCC Section 2-316. However, all this can be altered by contract. So an end user has incentives to avoid the formation of a license contract.

If you read a typical end user license, you will find many or most of these activities are "permitted." Of course, they need not be permitted because the copyright law says they are not violations of copyright in the first place. Remember that a license is no more than the promise not to sue for intellectual property infringement. Most end user licenses prohibit reverse engineering, disclaim liability of the licensor, and limit the warranties enjoyed by the licensee. Therefore, an end user will be in a worse position if he accepts the contract.

But those who want to exercise rights that go beyond mere use are in a different position. There is no background right to distribute or modify software. So, if a recipient of software wants to enjoy these rights, he must receive a license. This is why the formation wars have not materialized for open source, the way they did for EULAs in the 1990s. As we have seen, the licensor has little need for contract claims, and the licensee needs the license.

However, it would be unwise to assume that contract remedies are never available under open source licensees. In fact, lawsuits for

[12] *Sega v. Accolade*, 977 F.2d 1510 (9th Cir. 1992).

violations of open source licenses are often brought—rightly or wrongly—as contract claims.[13]

Of course, asking whether the GPL is a license or a contract assumes that it must be one or the other. Clearly, it is a license, but it may or may not be a contract. A written document is not a contract by itself; the document may contain the terms of a contract, but a contract is formed by actions. Whether one signs on the dotted line, clicks "I accept," or downloads software, a contract is formed with offer, acceptance, and consideration—and open source licensing does not change that.

Eradicating the Virus

I have long been on alert to find any formal legal underpinnings for the virus model. The only ones I have found are *Pickett v. Prince,* 207 F. 3d 402 (7th Cir. 2000) and *Anderson v. Stallone*, 11 USPQ2D 1161 (C.D. Cal. 1989). Both of these cases discuss the principle that an unauthorized derivative work is not entitled to copyright protection.

But these cases both involve very different facts from those of the typical GPL compliance problem. First, in each of these cases, the infringing derivative work probably did not meet a threshold level of creativity necessary for legal protection. The law sets this threshold very low. Any proprietary code with any commercial value would easily meet this threshold. Second, in each of these cases, the putative owner of the infringing derivative work was trying to sue the author of the infringed underlying work—something that requires a fair amount of good old-fashioned chutzpah. These facts cannot reasonably be extrapolated to the situation where the developer of a proprietary software module, which has been inappropriately combined with a GPL software module, tries to enforce its rights in the proprietary code standing alone, against a third party. The analog would be where a proprietary developer takes

[13] See *Progress Software Corp. v. MySQL AB,* 195 F. Supp. 2d 328 (D. Mass. 2002); and *MontaVista Software, Inc. v. Lineo, Inc.*, No. 2:02 CV-0309J (D. Utah filed July 23, 2002). See also *Artifex v. Hancom,* discussed in Chapter 19.

a blob of GPL code, changes one line of it, and then sues the GPL author for copyright infringement—which would be nonsense.

Finally, these cases, even if they stood for the principle that an unauthorized derivative work is not entitled to copyright protection, would indicate that the proprietary work would be in the public domain, rather than subject to GPL. That is a very scary result indeed—which fortunately is simply not supported by law.

Chapter 6

What Is Distribution?[1]

The conditions of most open source licenses—requirements to deliver notices, make source code available, or redistribute only on the same terms—are triggered by distribution. For almost all open source licenses, if you don't redistribute the software, you need not meet any conditions to exercise the license. But what is distribution? Twenty years ago the answer to this question was easy, but it gets more difficult every year.

An American Term of Art

The GPL is in essence a conditional copyright license, and it has no choice-of-law provision. Therefore, theoretically, only an action regulated by the licensee's local copyright law can trigger application of its copyleft conditions. In the United States, the core commercial right of copyright is called *distribution* or *publication*. Therefore, in the United States, the question of what triggers copyleft conditions is identical to the question of what constitutes distribution under copyright law.

GPL 3, which was released in 2007, attempted to internationalize its terms to fit with local variations on the distribution concept by using neutral words such as *propagate* and *convey*. Unlike its successor, GPL 2 specifically named distribution as the trigger for copyleft requirements. GPL 2 remains in wide use—and in particular it is the license that still

[1] An earlier version of this chapter appears in "The Gift that Keeps on Giving: Distribution and Copyleft in Open Source Software Licenses," IFOSSLR Volume 4, Issue 1 (March 2012). It originally appeared online at www.ifosslr.org.

applies to the Linux kernel—so the question of what constitutes distribution under GPL 2 is still alive and well in the open source world.

Distribution, though one of the enumerated rights of copyright under US law, is not defined in the Copyright Act (Title 17 of the United States Code). Title 17 grants a copyright owner the exclusive right to "distribute copies ... of the copyrighted work to the public by sale or other transfer of ownership, or by rental, lease, or lending."[2] The Act states that "offering to distribute copies ... to a group of persons for purposes of further distribution, public performance, or public display, constitutes publication."[3] But this does not define *distribution*. Where a statute's terms are ambiguous on their face, the rules of statutory interpretation allow us to look to the statute's legislative history. The 1976 House Report, a legislative rationale document produced by Congress in support of the relevant copyright statute,[4] also does not define *distribution*, but defines *publication* in the negative by saying, "[A]ny form of dissemination in which a material object does not change hands—performances or displays on television, for example—is not publication."[5] Later case law equated distribution with publication.[6]

Section 106(3) of the Copyright Act accords to the copyright owner the exclusive right "to distribute copies or phonorecords of the copyrighted work to the public by sale or other transfer of ownership, or by rental, lease, or lending." Put differently, the copyright owner has the exclusive right to publicly sell, give away, rent, or lend any material embodiment of his work.[7] As the legislative history of this section shows, the definition of *distribution* is "virtually identical with that in the definition of *publication* in section 101."[8] Thus, in essence, exclusive right of distribution is a right to control the work's publication.

[2] 17 U.S.C. Section 106(3).
[3] 17 U.S.C. Section 101.
[4] H.R. Rep. No. 94-1476.
[5] See H.R. Rep. No. 94-1476at 138, reprinted in 1976 USCCAN 5754.
[6] *Harper & Row Publs., Inc. v. Nation Enters.*, 471 U.S. 539, 552 (1985).
[7] *National Car Rental Sys., Inc. v. Computer Assocs. Int'l, Inc.*, 991 F.2d 426, 430 (8th Cir. 1993).
[8] Reg. Supp. Rep., p. 19.

Determining When Distribution Has Occurred

In the United States, therefore, *distribution* means providing a *tangible copy* to another person. The question of what constitutes distribution therefore devolves into two questions: What is a tangible copy, and what is *another person*?

The transfer of the work must be made "to the public" to trigger the definition of distribution under the Copyright Act. In the absence of a statutory definition of the phrase *to the public*, courts have held that a "limited" distribution that "communicates the contents of a manuscript to a definitely selected group and for a limited purpose, and without the right of diffusion, reproduction, distribution or sale," is not distribution to the public.[9]

In other words, a distribution is a "general" publication if it is not made (1) to a limited group; (2) for a limited purpose; and (3) "without the right of diffusion, reproduction, distribution or sale." The legislative history of the Copyright Act makes it clear that, "when copies or phonorecords are offered to a group of wholesalers, broadcasters, motion picture theaters, etc., publication takes place if the purpose is further distribution, public performance, or public display."[10] Thus, even if the work is distributed to a single person or entity, the publication would be general if the recipient is free to diffuse, reproduce, distribute, or sell copies of the work.

In the contemporary world of information technology, many activities stray close enough to the transfer of a copy to challenge the boundaries of this definition of distribution. It is these activities that make the question of what is distribution under GPL of such great interest to companies implementing day-to-day strategies for GPL compliance.

Starting at the baseline, the most obvious business case is that of a distributed on-premises product. Whether the product is software

[9] *White v. Kimmell*, 94 F. Supp. 502, 505 (S.D. Cal. 1950); *Data Cash Sys., Inc. v. JS&A Group, Inc.*, 628 F.2d 1038, 1042-43 (7th Cir. 1980) (concluding that "a 'limited publication' is really in the eyes of the law no publication at all").

[10] H.R. Rep. No. 94-1476, at 138 (1976).

alone or a hardware product as well, businesspeople understand what it means to sell a product and for it to change hands. Companies trying to comply with open source licenses like GPL 2 therefore have more difficulty assessing activities that they do not consider to be the business case of commercial distribution but that may nevertheless constitute distribution under the law. This chapter discusses those other business cases, from the clearest to the murkiest, as a matter of law.

A Clear Case in the Clouds

Companies often wonder whether software transmissions or remote use—sometimes called the SaaS model, or cloud computing—constitute distribution.

While this is one of the most controversial aspects of free software licensing, it is not a difficult interpretation question under US law for GPL 2. Advocates of free software have long recognized that if the trigger for copyleft requirements is distribution, increasingly popular cloud computing models will circumvent those requirements. This is sometimes referred to as the "SaaS loophole."[11]

During the drafting of GPL version 3, this issue engendered significant controversy. At one point, a variation on GPL 3 was proposed to allow the author to select an option that would cause online use to trigger copyleft requirements. Ultimately, this variation was removed from GPL 3 and memorialized in an alternative form of the license known as the Affero GPL. The basic form of GPL 3 makes it clear that ASP or SaaS use does not trigger copyleft requirements. In GPL 3, copyleft is triggered by *conveying* rather than by distribution, and "[t]o 'convey' a work means any kind of propagation that enables other parties to make or receive copies. Mere interaction with a user

[11] The term "ASP Loophole" (ASP here referring to an outdated term, popular in the 1990s and early 2000s, for what is now called SaaS) is often attributed to Richard Stallman, but that may not be accurate. See the April 3, 2007, interview with Mr. Stallman in Groklaw, in which he says the term is misleading: www.groklaw.net/articlebasic.php?story=20070403114157109.

through a computer network, with no transfer of a copy, is not conveying."[12]

Under US law, distribution requires actual transfer of a copy, in whatever form. Therefore, under US law, SaaS use—which involves accessing software without transfer of a local copy to the user—does not trigger copyleft requirements under GPL.[13]

The Edge Cases

Leaving aside the two relatively clear business cases of a distributed product (which clearly constitutes distribution) and pure SaaS deployment (which does not), we turn to some of the edge cases that are also common business activities but do not fall so neatly on one side of the distribution coin or the other.

- **Employees.** While companies often worry about this case, it is not a difficult one. Clients often ask whether "internal distribution" within a corporation triggers copyleft requirements. However, under law, there is no such thing as internal distribution because corporations and their employees are considered a single legal person. Therefore, one employee's providing a copy of software to another employee of the same company in the course of performing their duties as employees is clearly not distribution; while it may be a transfer of a copy, it is not a transfer to another person. Free software advocates sometimes refer to this as providing *private copies*.
- **Independent contractors—individuals.** Companies often engage individuals as independent contractors rather than as employees. Emerging growth companies, in particular, do this

[12] GPL 3, Section 0, Definitions, www.gnu.org/copyleft/gpl.html.

[13] It is worth considering that even in SaaS implementations, some components may be distributed. Today, most SaaS is accomplished only via a browser, so client software is no longer a common requirement to use SaaS. However, there are always exceptions, mostly notably JavaScript or mobile applications. Keep in mind that these are usually clearly distributed and would be subject to copyleft requirements.

to avoid the regulatory overhead costs (such as employment taxes) associated with hiring employees. The function of a contractor in such cases is nearly identical to that of an employee; however, because a contractor is not an employee, providing a copy of software to a contractor could be considered distribution. This is one of the thornier areas of GPL 2 interpretation, and it is discussed in more detail below.

- **Independent contractors—consulting firms**. Companies often hire small consulting firms to develop, test, or support software. These consultant entities often consist of a few persons working in a team, but their functional relationship to the company is similar to that of an individual consultant or an employee. Individuals in small consulting firms are not, legally speaking, employees of the company, and therefore, providing a copy to them is probably distribution. However, there may be arguments that the copies are not intended for public availability and that, thus, transferring them is not publication and therefore not distribution. This argument has risks, but it is probably supportable under law, particularly if buttressed by a written consulting agreement that recites the parties' intentions. This business case is very similar, in a legal sense, to the engagement of an individual contractor.
- **Independent contractors—outsourcing.** Larger companies often outsource entire business areas such as software development or software support. Outsourcers are clearly separate companies, rather than employees, so providing a copy to them is clearly providing a copy to a person other than the company. However, some outsourcing companies provide "leased" staff to work on servers and equipment owned or controlled by their customers. In this case, IT companies may reasonably make arguments that copies made available to those persons have not been transferred outside the companies' control. This argument may be less successful, however, for outsourcers who are outside the United States—as most are.

The international divide may make it unclear which body of law will determine what is distribution under GPL.

- **Cloud providers.** With the rise of cloud-based computing services such as Amazon Web Services, companies worry about whether uploading software to a cloud service is distribution. Although the question is subject to some debate, the answer is: probably not. Although doing this literally involves the transfer of a copy to the cloud service provider's computer equipment, the virtual space in which the copy resides is under the control of the company user; cloud services agreements generally do not provide the cloud service provider with any control over the information stored in a cloud account. Any transfer of a copy, then, is not "general" under copyright law.
- **Subsidiaries and affiliates.** Companies often create affiliate structures to conduct business for various strategic reasons such as tax planning, needing to do business in other countries through local entities, or engaging in a particular line of business. For example, a company may use a copy of the Linux kernel, which it has modified for its own purposes, to run an online service. It may provide this modified kernel to a subsidiary or affiliate in Europe or China to offer a local service. For tax, regulatory, or other reasons, it may be important to locate the servers for the business in Europe or China in those territories. If the recipient entity is a wholly-owned subsidiary of the company, the company has a good argument that, due to unity of ownership, the copy is a *private copy* that has only been given to the company itself and that, therefore, no distribution has taken place. This argument is also reasonably strong for a majority-owned affiliate because the parent effectively exercises control over the affiliate.[14] But if the recipient is a minority-owned affiliate, the company faces a more serious concern over

[14] In addition, because the copyleft requirements of GPL only allow binary recipients to seek source code copies, where the recipient is a majority-owned affiliate, the issue may be moot; the recipient would simply never make the request.

whether distribution has taken place. This scenario is quite common, particularly where companies have little choice but to create minority-owned operating entities in territories, like India or China, that impose significant restrictions on foreign ownership of businesses operating within their borders.

- **Mergers and acquisitions.** US law can be quirky and counterintuitive on the subject of assignments by operation of law in connection with mergers and acquisitions. An assignment of a contract (or a license) occurs when one party to the contract transfers its rights to another. Therefore, for instance, if a corporation enters into an agreement with another party, it may be able to transfer that agreement to another corporation—depending on what the agreement has to say about it. Contracts are generally considered assignable under US law,[15] but intellectual property licenses are subject to different rules. Generally, nonexclusive copyright and patent licenses are not assignable.[16] Therefore, if a corporation takes a nonexclusive license to a patent, it cannot transfer the license to another corporation unless the license agreement expressly allows transfer. To make matters even more complicated, there are courts that have held that an acquisition—even a transaction such as a reverse triangular merger in which the target entity survives—can be an assignment by operation of law. Even if the licensee is the same corporation before and after the acquisition, the license may not be exercisable after the transaction. This rule of law may also have implications for the

[15] Other than special kinds of contracts, where assignment would change the basic nature of the contract, like contracts for personal services or requirements contracts. See Restatement (Second) Contracts, Section 317.

[16] For patent, see *PPG Indus. Inv. v. Guardian Indus. Corp.*, 597 F.2d 1090 (6th Cir. 1979). For copyright, although the law is conflicting see, e.g., *SQL Solutions, Inc. v. Oracle Corp.*, 1991 U.S. Dist. LEXIS 21097 (N.D. Cal. 1991). This is an unpublished decision and arguably contrary to the California Supreme Court's view in *Trubowich v. Riverbank Canning Co.*, 182 P.2d 182 (Cal. 1947).

definition of distribution. If a change of control is an assignment by operation of law, one might logically conclude that it also constitutes providing a copy to another entity and thus a distribution triggering copyleft obligations. Keep in mind, also, that the effectuation of some forms of M&A transactions, such as asset sales, are clearly assignments and also likely to constitute distribution under GPL 2.

- **Productization.** Although this business case is not complex from a legal standpoint, it is such a frequent trap for companies managing open source compliance that it is worth mentioning in any discussion of distribution issues. Companies that offer SaaS solutions tend to rely on the fact that they are not distributing their products to ensure their GPL compliance. They do this by merely avoiding licenses like Affero GPL that have requirements even in the absence of distribution. However, this can be a dangerous strategy. For a business development manager who is not focused on legal and technical niceties, it is easy to cause transactions to trip over the distribution line. A company with a SaaS offering may, for instance, approach a customer operating in a highly regulated market (such as health care or financial services) that will insist that the SaaS offering be operated via a private instance on the customer's premises or on servers under the customer's control. This demand usually arises from security or regulatory auditing concerns. From the business point of view, a private instance of a SaaS product is a technical detail. But of course, providing a copy to the customer will likely constitute distribution. If the company's open source compliance strategy hinges on refraining from distribution within the context of a SaaS model, the company may find that it cannot deliver a compliant product in any reasonable amount of time—usually because it has intermixed GPL- and non-GPL-compatible code or has not properly kept track of open source elements in the product.

With these edge cases in mind, we now turn to extrinsic evidence of the meaning of GPL 2 and best practices in managing distribution issues.

The FSF View

The GPL 2 FAQ, promulgated by the Free Software Foundation (FSF), offers the FSF's insight into what it considers a distribution that would trigger copyleft requirements. For example, one of the FAQs is as follows:

> Is making and using multiple copies within one organization or company "distribution"?
>
> No, in that case the organization is just making the copies for itself. As a consequence, a company or other organization can develop a modified version and install that version through its own facilities, without giving the staff permission to release that modified version to outsiders.
>
> However, when the organization transfers copies to other organizations or individuals, that is distribution. *In particular, providing copies to contractors for use off-site is distribution* [emphasis added][17]

The FAQ also discusses a transfer between an organization and a majority-owned subsidiary:

> Does moving a copy to a majority-owned, and controlled, subsidiary constitute distribution?
>
> Whether moving a copy to or from this subsidiary constitutes "distribution" is a matter to be decided in each case under the copyright law of the appropriate jurisdiction. The GPL does not and cannot override local laws. US copyright law is not entirely clear on the point, *but appears not to consider this distribution* [emphasis added].
>
> If, in some country, this is considered distribution, and the subsidiary must receive the right to redistribute the program, that will not make a practical difference. The subsidiary is controlled by the parent company;

[17] This can be found at: www.gnu.org/licenses/old-licenses/gpl-2.0-faq.html#InternalDistribution. This same FAQ appears in the GPLv3 FAQ as well (www.gnu.org/licenses/gpl-faq.html#InternalDistribution).

> rights or no rights, it won't redistribute the program unless the parent company decides to do so.[18]

In this FAQ, the FSF acknowledges that, at least in the United States, a transfer to or from a majority-owned and majority-controlled subsidiary may not constitute distribution. Further, the FSF gives weight to one organization's effective control over another to determine whether the two entities are effectively one entity for the purposes of the analysis.

There is also discussion in the GPL 2 FAQ about providing modifications of GPL code under a nondisclosure agreement:

> Does the GPL allow me to develop a modified version under a nondisclosure agreement?
>
> Yes. For instance, you can accept a contract to develop changes and agree not to release your changes until the client says OK. This is permitted because in this case no GPL-covered code is being distributed under an NDA.
>
> You can also release your changes to the client under the GPL but agree not to release them to anyone else unless the client says OK. In this case, too, no GPL-covered code is being distributed under an NDA, or under any additional restrictions.
>
> The GPL would give the client the right to redistribute your version. In this scenario, the client will probably choose not to exercise that right, but does have the right.[19]

Many companies find the distribution question confusing because they find this FAQ confusing. In this FAQ, the FSF considers two different scenarios: (1) the contractor releases the modified code to the public generally at the direction of the client, and (2) the contractor releases the modified code to the client under the GPL, and the

[18] This can be found at: www.gnu.org/licenses/old-licenses/gpl-2.0-faq.html#DistributeSubsidiary.
This same FAQ also appears in the GPLv3 FAQ as well at the online address: (www.gnu.org/licenses/gpl-faq.html#DistributeSubsidiary).

[19] This can be found at: www.gnu.org/licenses/old-licenses/gpl-2.0-faq.html#DevelopChangesUnderNDA.
This same FAQ also appears in the GPLv3 FAQ as well at the online address: (www.gnu.org/licenses/gpl-faq.html#DevelopChangesUnderNDA).

contractor promises not to release the modified code to anyone else. Unfortunately, this FAQ section does not specify whether "a modified version" refers to a modification of the contractor's own GPL code, to GPL code that may have been already modified by the client, or to a modification of third-party code. Clearly, these three situations could be analyzed differently. If the FAQ refers to GPL code owned by either the client or the contractor, it is a trivial question; obviously, the owner of GPL code can choose to deliver that code under GPL terms or not as it sees fit, because an author (as licensor) is not bound by the copyleft obligations of GPL—only the licensee is. If the FAQ refers to modifications to third-party code, it implies that even if the delivery of the original code constitutes a distribution, that distribution does not trigger the copyleft obligations of GPL.

Other information promulgated by the FSF suggests that this FAQ element is not intended to address third-party code. But that is, by far, the most common situation: a company wants to use some GPL code but needs modifications, so it finds an expert in the code willing to modify it on a contract basis. In fact, this scenario is so common that it is touted as one of the advantages of open source software. But the company may not plan to ever distribute the software. Therefore, if providing the code to the consultant is distribution that triggers copyleft requirements, the company will likely be unwilling to engage the consultant.

The FSF's view is problematic for a couple of reasons. First, a practical problem: companies that hire consultants simply don't distinguish between the business cases of in-house and contractor development. They do not expect to encounter a completely different GPL compliance landscape based on the distinction. Because FSF's view contravenes business expectations, it is a trap for the unwary. Second, a legal problem: the provision of code for development purposes is more akin to "communicat[ing] the contents of a manuscript to a definitely selected group and for a limited purpose, and without the right of diffusion, reproduction, distribution or sale" (i.e., not publication under copyright law) than it is to common notions of

redistribution or publication. Therefore, there is a strong argument that such a transfer is not distribution under the law.

The International View

It is important to keep in mind that the distribution question as it is analyzed here is largely unique to US law. Because GPL 2 does not have a choice-of-law provision and is a conditional copyright license, it governs only what is protected via local copyright law. A full discussion of the tenets of international copyright law bearing upon this issue is beyond the scope of this chapter, but it seems likely that the question could have different answers outside the United States. The Berne Convention for the Protection of Literary and Artistic Works, as amplified by the WIPO Copyright Treaty, provides for a right to "make available" a literary work. This may be broader than the US notion of distribution, and most importantly, it could include SaaS offerings. Therefore, the triggers for copyleft obligations based on activity outside the United States may have a lower threshold than those based on activity in the United States.

Best Practices for Contract Drafting and Deal Structuring

As lawyers in private practice await clarity in the common law on distribution issues, they may implement drafting and structuring practices to clarify their clients' intent or to minimize the uncertainty of the results should the courts later announce decisions on distribution questions. Such practices cannot address all distribution issues should there be a contrary court pronouncement, but they might help discourage claims, provide evidence of intent, or reduce confusion when those not directly involved in the deal are later asked to assess distribution issues.

Development Agreements

To avoid confusion about whether development activities constitute distribution, consider adding language such as the following to your development agreements:

- *Contractor shall conduct development services only on systems and equipment under the control of Customer.* This language limits work to customer-controlled servers. This will address whether a distribution has occurred, the theory being that even though the contractor is a separate person, no copy of the software has been transferred.
- *Contractor acknowledges that it is performing the development services solely for the benefit of Customer, and solely as directed by Customer, and shall not make any copy of the Software available to any other person or entity.* This language states that copies are intended to be private, thus addressing the situation that the FSF FAQ says constitutes distribution.

These approaches are attractive because they comport with customary confidentiality provisions and "work made for hire" provisions in development agreements, which often recite customer control of the development activities to support treatment of the work as "work made for hire" under *CCNV v. Reid*.[20]

[20] *Community for Creative Non-Violence v. Reid*, 490 U.S. 730 (1989), held that the factors for determining whether a work of authorship is a work made for hire (owned by the company) or not (owned not by the company but by the author) are, among others, the level of skill required to create the work, the source of the tools used in creating the work, where the work was created, the duration of the relationship between customer and author, the extent of the contractor's discretion over when and how long to work, and whether the work is part of the regular business of the customer or consultant. Therefore, many consulting agreements recite where work will be performed, as well as other facts that might bear on whether distribution has occurred.

Mergers and Acquisitions

- Avoid delivery of GPL software. Particularly in asset purchase deals, determine whether there is a reasonable way to refrain from delivery of open source packages in favor of having the buyer download them directly from the original source or a third-party source. This approach is useful mostly in situations in which drivers or other significant original code belonging to the seller is being delivered. It is not useful when the seller is delivering integrated modifications. In that case, the seller would deliver only its additions, and the buyer would receive third-party open source code separately. Clearly, if third-party open source code is extensively modified, this strategy may not be feasible because it would be so difficult to separate the seller's code from the third party's code. However, companies that are very conservative on this issue may deliver only *diffs*, or patches, in an attempt to avoid delivery of any third-party GPL code. Keep in mind that distribution is usually an issue for the seller, not the buyer. Therefore, asset purchases that consist of all the assets of the seller entity may render the concern moot, but a seller's divestiture of partial assets, business lines, or product lines may cause the seller to have concerns about GPL distribution. A seller wishing to sell its own code may find buyers unwilling to pay for that code if the code must be delivered under GPL.

SaaS Agreements

- Avoid drafting that confuses SaaS with distribution. SaaS agreements are not primarily licenses; they are service agreements. Sometimes, as an artifact of their business antecedents in distributed software, SaaS agreements are drafted so much like distributed software licenses that it is difficult to tell the two apart. Although the distribution question would likely turn primarily on the supplier's actions, not merely on document drafting, it is best not to hurt your

position by using a SaaS agreement that reads like it covers a distributed product.

Intercompany Agreements

- Recite intent not to distribute. In software agreements between corporate affiliates, parent entities may wish to clarify that no distribution is intended, much in the same way as recommended above for consulting or development agreements. This may seem obvious, but in fact, intercompany technology licenses are often not drafted by technology lawyers. Instead, they are drafted by tax lawyers or corporate lawyers who are documenting intercompany arrangements for the purpose of managing imputed tax issues, rather than precisely considering intellectual property issues. It is crucial to review these agreements with a view to open source as well as intellectual property issues.

An Enduring Puzzle

It is unlikely that the federal courts in the United States will answer these distribution questions anytime soon. The open source enforcement actions that have been brought to date have not seriously addressed these questions. Given that other heady issues (such as the scope of derivative works under GPL 2 and the interaction of patent law and open source licensing) are still unclear in open source law, they may not be ripe for dispute. Also, most authors who release code under GPL 2 are simply not focused on issues like intercompany agreements and mergers or acquisitions. This is because they are primarily technologists rather than corporate strategists. If GPL authors generally do not intend to enforce their rights in these edge cases, there may not be a constituency that is interested in bringing a lawsuit that will make law in this area. It therefore seems likely these questions will persist as long as GPL 2 remains a widely-used license, and based on the prevalence of the Linux kernel alone this will be a long time. Companies assessing open source compliance should be sure they have identified the types of

distribution that are most likely to be questioned so they can use open source software with confidence and plan their transactions in a way that comports with their open source compliance strategy.

Chapter 7

Notice Requirements

Notice and attribution requirements constitute one of the least intellectually challenging but most important parts of open source licensing. All open source licenses—from the most permissive to the strongest copyleft—require licensing notices. On their face, these requirements are straightforward, but anyone who has tried to create a notice file for a binary product can attest that the task can be challenging and frustrating. How, they think, can such a simple task be so difficult? But one thing is sure: if a redistributor fails to use proper licensing notices, it creates the simplest path for an open source licensor to claim and prove noncompliance. In fact, most litigation concerning open source enforcement is based on failure to provide notices.

What Is a License Notice?

The requirement to deliver notices is merely that—a requirement to deliver a text file informing the recipient that certain open source software, which is available under the noticed license, is included in the software being delivered to the recipient. Sometimes, that notice also acts as the operative licensing terms, but not always.

For example, if a product is distributed under an end user license agreement and contains some elements of third-party open source software that are available under the BSD license, the licensor must give a copy of the BSD license to the recipient of the software. However, the software as a whole is not licensed under BSD. Effectively, this license notice serves to inform the recipient that the recipient is receiving a license to that element directly from the licensor under BSD, even if the larger software is not available to the recipient under BSD. Of course,

permissive licenses do not have source code delivery requirements, so the recipient may not know where to get a copy of the source code. Nevertheless, the recipient may choose to look for a copy independently.

For copyleft licenses, the license notice serves to inform the recipient that the software is provided under those license terms, directly from the licensor—or in the case of the weak copyleft licenses like Mozilla Public License or Eclipse Public License, that the source code is available under those terms. In the case of licenses like GPL and LGPL, binaries must be provided only under those terms, so the licenses serve as the operative licensing terms for the software.

Most licenses also include a copyright notice, such as Copyright 2019 XYZ, Inc. The copyright notice is usually at the top of the license, but some open source authors omit the copyright notice. The license notice requires one to reproduce what was provided—no more, no less.

How to Create a License Notice

Creating the exact form of a license notice can also be challenging. Almost all open source licenses only require the text file of the license to be delivered. Typically, the license notice is delivered in plain text format.

Different licenses require delivery of notices in different circumstances. Here are some examples:

- **MIT.** "The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software."
- **BSD.** "Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution."
- **GPL 2.** "You may copy and distribute verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on

each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and give any other recipients of the Program a copy of this License along with the Program."

- **Apache 2.** "(1) You must give any other recipients of the Work or Derivative Works a copy of this License ... (3) You must retain, in the Source form of any Derivative Works that You distribute, all copyright, patent, trademark, and attribution notices from the Source form of the Work, excluding those notices that do not pertain to any part of the Derivative Works; and (4) If the Work includes a "NOTICE" text file as part of its distribution, then any Derivative Works that You distribute must include a readable copy of the attribution notices contained within such NOTICE file, excluding those notices that do not pertain to any part of the Derivative Works, in at least one of the following places: within a NOTICE text file distributed as part of the Derivative Works; within the Source form or documentation, if provided along with the Derivative Works; or, within a display generated by the Derivative Works, if and wherever such third-party notices normally appear. The contents of the NOTICE file are for informational purposes only and do not modify the License. You may add Your own attribution notices within Derivative Works that You distribute, alongside or as an addendum to the NOTICE text from the Work, provided that such additional attribution notices cannot be construed as modifying the License."

There are variations of these requirements. For example, some forms of the BSD license do not require delivery of a license notice with binaries, only with source code.

However, because a product often has many open source components, it is unmanageable to have a different license notice process for every license. Therefore, most companies seeking to comply with open source notice requirements develop an internal process for

preparing notices that is consistent with the most stringent common notice requirements; GPL is usually the model.

The notice requirement almost always requires delivery of an entire copy of the license. Some licenses allow a form of abbreviated notice. But again, it is not practical to implement different internal processes for different licenses, so most companies provide a copy of the entire license for every open source license, whether it is required or not.

If a software product is delivered in source code form, the notices are usually "baked in." If a developer downloads an open source component for his product, the license file is usually contained in a text file called *license.txt* or *copying.txt*, which is included in the download package along with the source code files for the component. If the developer simply redistributes all of the source code files along with the product, there is often no need to create a separate license notice. This is the best and easiest approach.

However, many companies do not want to deliver source code up front. There are many reasons for this. Often, it is impractical to do so. In some cases, the company objects to the requirement to deliver the source code when the licenses don't require them to, such as with permissive licenses like BSD or MIT. When the license imposes no condition to deliver source code, doing so is a choice, not a requirement. In the case of copyleft licenses like GPL and LGPL and weak copyleft licenses such as Mozilla Public License, there is no requirement to deliver source code with every binary. However, there is a requirement to inform the recipient that the source code is available under the terms of that copyleft license. In such cases, the developer has a choice whether to include the source code up front with binaries or not, but when he elects not to do so, he must prepare a separate notice file. It is creating this extra file of licensing notices that causes companies so much distress.

Accordingly, the easiest and best practice is always to deliver source code up front. Not only does this ease or eliminate the task of preparing licensing notices, but it also will deflect requests for source code that must be made available under copyleft licenses, the failure to complete which results in noncompliance.

If the product is a hardware product, and if the software is embedded or has no user interface, then delivery of notices can be particularly difficult. Companies continually insist that they should be able to deliver notices via the Internet instead of with a product distribution. Unfortunately, although that might be a convenient solution, it does not comply with the notice requirements of most open source licenses. The open source license almost always requires that an entire copy of the license be delivered with the product. If the product is a software product, this usually means the notice files are in the product download or on the distribution media (such as CD-ROM). If the product is a hardware product, there may be no user interaction screen suitable for displaying the notices, and even if there is a screen, a small screen will make reading the notices unpleasant. Ironically, this is usually a compliant approach, although it may not serve the user in a practical sense. If you have a smartphone, for instance, you can probably easily find the open source license notices in it. However, you will find reading hundreds of pages of notices on a very small screen extremely difficult. Companies often argue—not unreasonably—that it would be better to point the user to any web page where notices could be provided in a more digestible format. Unfortunately, this is not what most of the licenses require.

The theory behind disallowing online notices and requiring reproduction of the entire license (even though open source licenses are almost always readily available online) is that when most of the licenses were originally written, most people did not have Internet access. Of course, in the twenty-first century, Internet access is ubiquitous. So the reason for the original requirement to deliver the entire copy of the license may be anachronistic, but open source license terms do not change. Therefore, for many years in the future, much legacy code will be distributed under licenses that require notices to be delivered in ways that do not take advantage of modern technology.

There is one common circumstance in which it is reasonable—and probably compliant—to deliver licensing notices via the Internet, and that is when delivering licensing notices for an Internet product. For example, JavaScript—a language that executes in the user's browser in

source code form—is often distributed in connection with SaaS and web services. Much JavaScript is under open source licenses. When pushing JavaScript out via a web browser, it is reasonable to provide a link where the user can go to find licensing notices. In such a case, the JavaScript could not possibly be distributed to the user unless the user had access to the Internet to follow the link. Though this may or may not comply with the letter of some open source notice requirements, it complies with the spirit.

A similar edge case might exist for a device that only works with an Internet connection—particularly a device that has no user interface. In this case, delivering notices via a web link would be an alternative to delivering a set of paper notices or notices on electronic media such as a CD-ROM—both which are expensive to do.

Finally, once a company decides on its procedures for delivering notices, it must take care to keep them current. Notices must be updated as open source components of the products change over time. Unfortunately, outdated license notices are very common.

Attribution and Advertising Requirements

Some open source licenses have more troublesome notice requirements, such as requirements to deliver notices "in the user manual"—which made sense 20 years ago but not nowadays, when the creation of separate user manuals is becoming less common every year. In addition, a certain set of licenses contained what are called advertising requirements. Apache 1.0 Section 3 contained this provision:

> The end-user documentation included with the redistribution, if any, must include the following acknowledgment: "This product includes software developed by the Apache Software Foundation (http://www.apache.org/). Alternately, this acknowledgment may appear in the software itself, if and wherever such third-party acknowledgments normally appear."[1]

[1] www.apache.org/licenses/LICENSE-1.1.

This kind of "advertising requirement" has been deprecated, not only because it is costly and difficult to implement but because it is incompatible with GPL.

One popular piece of software that remained for many years under such a license is OpenSSL, whose license says this in Section 3:

> All advertising materials mentioning features or use of this software must display the following acknowledgment:
>
> "This product includes software developed by the OpenSSL Project for use in the OpenSSL Toolkit. (http://www.openssl.org)"[2]

Requirements like this can be difficult to interpret. What constitutes advertising material? Does any product listing noting that the product is compatible with SSL meet this definition? What does it mean to mention "features?" In the case of a package like OpenSSL that implements a protocol (SSL) with other implementations, is this requirement triggered by mentioning SSL or only the features particular to the OpenSSL implementation? Provisions like this are unpopular—and not a best practice in open source licensing—because of their vagueness and quick obsolescence.

Noting Modifications

Some notice requirements of open source licenses are often forgotten. For instance, many open source licenses require the developer to note his own changes. For example, GPL version 2 says, "You must cause the modified files to carry prominent notices stating that you changed the files and the date of any change." In practice, this only requires a name and date, not a description of the changes.

Some other licenses have different and broader modification notice requirements. These requirements may seem burdensome on their face; they were historically considered good programming practice, regardless of the license requirement. Today, however, most information about changes to software is stored as metadata in a

2 www.openssl.org/source/license-openssl-ssleay.txt.

concurrent versioning system, rather than in source code comments. So including this as a notice requirement is not popular among engineers.

Automation

There is some good news in the grim task of notice creation. Clearly, it is an area ripe for automation. Some build systems (such as Debian Linux and the Android system) contain functionality that helps gather information for notice files. Audit reports generated by code scanners like Black Duck Software or FOSSA can often be culled for license notices.

Although not intended primarily to address notice delivery, the SPDX (Software Package Data Exchange) project also promises to assist in the automation of the delivery of licensing information. This project works under the aegis of the Linux Foundation. SPDX is a standardized format for communicating licensing information. SPDX contains interesting information about licensing that can be useful for generating or checking notice files for end users, but it is primarily intended to address the delivery of information in supply chains—from one developer to another. This is an issue that goes beyond notices or even the SPDX project. As it currently stands, business users of open source software waste much effort when creating information disclosures at every level of the supply chain. (For more on this, see Chapter 17: Mergers & Acquisitions and Other Transactions.)

David Marr eloquently describes this issue as follows:

> FOSS exists in all levels of a supply chain, from the first software developer creating software all the way to the final packaged product that is sold to end users. However, in the context of a commercial supply chain, the current FOSS ecosystem is broken. If a supply chain can be compared to a stream or a river, as software flows down the supply chain—i.e., when software is delivered from one company to the next—each successive downstream company is redoing portions of the compliance work already done (or what should have been done) by the

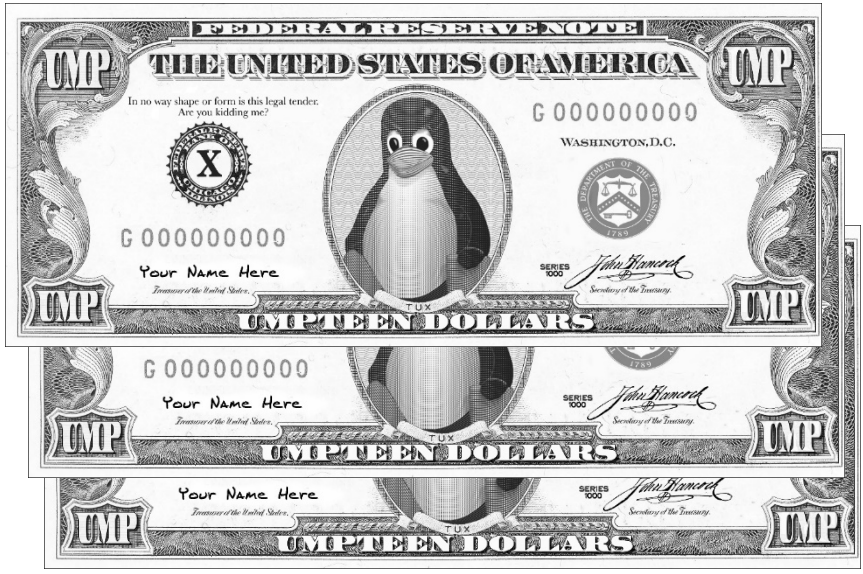
> upstream company. This is all done at unnecessary cost and inefficiency, and it often delays time-to-market.[3]

The challenge of conveying license notices and the challenge of conveying licensing information are, of course, related. License notices are required for all distribution—and that includes every intermediate distribution within the supply chain. So, every supplier in the chain has the obligation under the applicable open source license to convey license notices and the business necessity to deliver the substance of the licensing information to its customers. Although one is a license compliance issue and the other is an information management issue, the resolution to these issues should dovetail in a standardized and compliant process.

[3] Email, on file with author. Mr. Marr is a well-known open source legal expert and active participant in the SPDX project.

Part III

Advanced Compliance



Chapter 8

The GPL 2 Border Dispute

The single most difficult and controversial legal issue in open source licensing is the scope of GPL version 2. When clients ask about this issue, they usually use words like "tainting" or refer to the "viral" nature of GPL. But in an effort to use more neutral language, I call it the *border dispute* because the basic issue is this: What does GPL cover, and what does it not cover? In the language of the license, what is a *work based on the Program* versus something entirely separate that does not need to be covered by GPL?

The copyleft conditions of the GPL apply to the original work released under the license (the "Program") and works "based on the Program." But the GPL does not seek to control other works that are "mere aggregations" with the Program. In law school, we learned that any use of the word *mere* signaled that circular logic was about to follow, and analysis of the GPL is no different. There has been a fair amount written about the border dispute, but much of it is not well reasoned because it's advocacy rather than analysis. In analyzing issues of GPL's scope, it is necessary to know who says what about this topic, and why. But a reasoned discussion of this topic tests one's understanding of what principles guide the interpretation of licenses in a way that few other topics do.

Developers of proprietary software are extremely concerned with this issue—and rightly so, if they want to avoid breaching the GPL. Free software advocates have a tendency to dismiss the angst this topic generates in proprietary software developers. Some believe that developing proprietary software is morally wrong to begin with and that proprietary developers bring these concerns upon themselves—therefore, they believe that such concerns shouldn't trouble the pure of

heart. But in fact, we live in a world where open source and proprietary software must coexist. Also, many of the biggest open source projects today are supported largely by industry players who develop proprietary software. These players need to know where they stand—where to draw the line between what must be GPL and what need not—and they are often frustrated by the GPL's lack of clarity on this question. But perhaps more importantly, justice demands that the rules of a society be clear and understandable, or else only those with arcane knowledge can fully participate in society. GPL is a kind of constitution for the free software world, so its lack of clarity is a continuing concern for those working in that world—even those who want to promote free software.

Speaking from experience, I have almost never heard a client say it wanted to violate GPL, but I have heard hundreds of clients say they were confused about what is necessary to avoid violating GPL. Responding to this concern by suggesting that all these companies should not worry about the rules and simply release all their software under GPL is neither realistic nor the best result for open source software. Private enterprises usually could not do this if they wanted to; they owe a duty to their shareholders to make money, and due to inbound license limitations, they often do not have the right to release all their software under open source licenses. Those who feel this is morally wrong and that all software companies should be not-for-profit entities don't believe in the inventive value of private enterprise, and thus, they play into the stereotype that allowed anti-open source factions to get traction in the 1990s. In fact, most open source developers, proponents, and supporters are neutral about this topic and understand that open source is a great model for some software and not such a great model for other software. Just as judges have long pondered the balance of intellectual property protection and public domain that will best foster innovation, there is a balance of scope to open source licenses that will result in maximum contribution to open source by the technology industry. Lack of clarity in the rules of the road imposes a risk-avoidance tax on the technology industry, and private companies consider pondering GPL's scope to be one of the hidden elements of the total cost of ownership of open source software.

Libraries and Other Standard Elements

To understand how scope issues are analyzed, take the case of a humble but useful software routine: a time function. If an application wants to find the current time, it might make a call like this:

```
Time_t t=time(NULL);
```

Don't worry if the syntax looks odd to you. This code is doing something very simple. The program that uses this line of code is defining a variable called t that will be populated with the current time, using the function `time`. Once the program gets that information, it can display the time on the user interface screen, make calculations using the time, or do whatever else it wants to do with time information. Obviously, many application programs might need to use this basic function. To do so, the application needs to make the system call (`time`) using the library of code containing the time function, while the application program is running. Suppose you are writing a calendar application (the "Application")—clearly, you will need to know the current time and date and update it frequently while the application is running. Your calendar application will make frequent use of this library function time (the "Library"). Translating this into the context of the question of GPL scope, we would ask, "Under GPL, are the calendar Application and the time function Library two programs or one?"

Now, this is in fact a trick question. By answering this question, we will see the border dispute of GPL in action. In other words, we are seeking, based on the language of GPL, to determine which of the following is true:

- Case One: The Application and the Library are part of the same program and thus both must be covered by GPL.
- Case Two: The Application and the Library are different programs, and thus if one of them (the Library) is covered by GPL, the other (the Application) need not be covered by GPL.

What the GPL Says[1]

The language of GPL that governs this question centers on the phrase *work based on the Program*. The Program means the software released under GPL. GPL imposes conditions that govern the distribution of the Program or any work based on the Program.

Paragraph 0 says, "This License applies to any program or other work which contains a notice placed by the copyright holder saying it may be distributed under the terms of this General Public License. The 'Program,' below, refers to any such program or work, and a 'work based on the Program' means either the Program or any derivative work under copyright law: that is to say, a work containing the Program or a portion of it, either verbatim or with modifications and/or translated into another language."

Paragraph 2 says, "You may modify your copy or copies of the Program or any portion of it, **thus forming a work based on the Program.**"

Paragraph 2(b) says, "You must cause any work that you distribute or publish, that **in whole or in part contains or is derived from the Program or any part thereof**, to be licensed as a whole at no charge to all third parties under the terms of this License." This can be thought of as the copyleft provision of GPL, because it imposes the condition of relicensing under the same terms.

Paragraph 2 also says, "These requirements apply to the modified work as a whole. If identifiable sections of that work are **not derived from the Program, and can be reasonably considered independent and separate works in themselves**, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Program, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to **the entire whole, and thus to each and every part** regardless of who wrote it.

[1] In this section, all emphasis in quotations from the GPL has been added by the author.

"Thus, it is not the intent of this section to claim rights or contest your rights to **work written entirely by you;** rather, the intent is to exercise the right to control the distribution of **derivative or collective works based on the Program.** In addition, [this section] does not bring the other work under the scope of this License."

Paragraph 5 says: "You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Program **or its derivative works.**"

How Courts Interpret the Language of Contracts

At this point, do you know the answer to our "time" question? Of course not. The language of GPL requires interpretation.

When lawyers interpret documents such as contracts, statutes, or regulations, they use rules that are old and well settled. These rules seek to determine the intent of those who wrote the documents, at the time they were written. Though some claim the GPL is a license and not a contract, there is no reason to think that the rules of interpretation would be different from those applied anywhere else in the law.[2]

The most basic rule of contract interpretation is the *four corners rule*, which holds that one first tries to discern the meaning of the document from the language within its four corners. (Think of it as WYSIWYG). Another important rule is the *parol evidence rule*, which holds that if a written document exists, prior or contemporaneous oral statements are not evidence of intent. Also, where possible, a document is interpreted as a whole, giving effect to all parts of the document. We can't just pick and choose the parts we want to use; we have to assume that the whole document—every phrase of it—has meaning.

[2] Below, I apply these rules as articulated in the Restatement (Second) of Contracts and the Uniform Commercial Code. While a contract would normally have a law selection clause that applies the law of a specific US state, GPL does not have a governing law-selection clause. Licenses of software are generally considered to be covered by UCC Article 2, and those rules are consistent with contract common law on interpretational rules.

This leads us to an important idea. Interpretation of a document depends primarily on the objective meaning of the words of the document, not on what others think or say about them. Interpretation goes through a series of steps: First, the exact words in the document are assigned their ordinary, plain meaning (such as the meaning they would have in a dictionary). If the dictionary definition still leaves the meaning ambiguous, then we use other rules of contract construction to resolve the ambiguity.

Those rules look outside the four corners and dictionary definitions. For example, a meaning given to a word by a trade or profession will take precedence over the dictionary definition of the word, if that makes sense in context. In computer software licenses, obviously, technical meanings are very important. For example, GPL has this to say about source code:

> For an executable work, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the executable. However, as a special exception, the source code distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

Words like *kernel* and *scripts* obviously have special meanings here. They don't refer to wheat bran and movie dialogue.

Beyond this, there are two other interpretational rules that are often confused with each other: course of performance and usage of trade.[3] *Course of performance* (sometimes called course of dealing) describes how the parties to an agreement perform the agreement. For example, if you agree on paper to pay your rent on a monthly basis, but if the paper does not say what day of the month you will pay and you pay on the first of the month for six months, then the paper will be

[3] The Restatement describes *course of performance* as "repeated occasions for performance by either party with knowledge of the nature of the performance." *Usage of trade* is defined in the UCC as "any practice ... having such regularity of observance in a place, vocation, or trade as to justify an expectation that it will be observed with respect to the transaction in question."

interpreted to require you to pay on the first of the month because of the way you have performed the contract so far. *Usage of trade* describes how the industry generally operates. For instance, if you write a contract to buy a wooden beam that is a "finished two-by-four" and the supplier delivers to you a beam that is slightly smaller, you will not have a legal claim. It is customary in the trade to call such beams two-by-fours even though after finishing they are slightly smaller.

Finally, if the rules of contract construction fail to resolve the ambiguity, then under the rule of *contra proferentem*, any ambiguity must be construed against the drafter of the contract. This rule is meant to protect those who don't wield the drafter's pen.

Moreover, if a particular party to a license makes a statement against his own legal interest, then as a matter of fairness, the courts will not sustain legal claims against those who reasonably rely on that statement. For instance, suppose the author of the Library said publicly, "I have licensed this software under GPL, but I don't intend ever to enforce its terms against anyone in schools or nonprofit organizations." Then, any lawsuit by the author against a school or nonprofit for violating the GPL would probably not be successful. This principle is known as waiver or estoppel—the author has waived his rights, or as a matter of fairness, the author should be stopped from suing for infringement.

Applying the Four Corners Rule to GPL 2

Now we have a question, a set of language to interpret, and a set of rules to make our interpretation. This is what lawyers do, and even if you are not a lawyer, you can do it, too. You just need to think carefully and precisely about the language and think of all the meanings it might have.

The GPL is not a traditional legal document. If it were, the term *work based on the Program* would be capitalized whenever it is used and defined once, and only once, at the beginning of the document. When lawyers draft legal documents, this is how we seek to minimize the

ambiguity of language. It is also, of course, how ambiguities are avoided in programming languages: variables are defined at the beginning and should only be used in a consistent way, or there will be bugs. If the GPL's omission of definitions of important terms like *work based on the Program* seems ironic to you—well, it does to me, too.

Taking the language of GPL at face value, we would need to assess the boundary or overlap between, on the one hand, "works written entirely by" the licensee, works combined by "mere aggregation" on the same storage volume or medium, and works that "can be reasonably considered independent and separate works in themselves" and, on the other hand, a work "that in whole or in part contains or is derived from the Program or any part thereof." Keep in mind that we do not have the option of discarding any of these phrases if they don't seem necessary or useful to us. Every term in the document must have meaning.

This is why attorneys who read the GPL quickly come to the conclusion that the phrase "work based on the Program"—upon which entire companies and development projects depend—is irretrievably vague. To illustrate this point, Figure 8.1 provides a Venn diagram of the various definitional phrases of the language of GPL 2 that I drew once in a moment of philosophical contemplation.

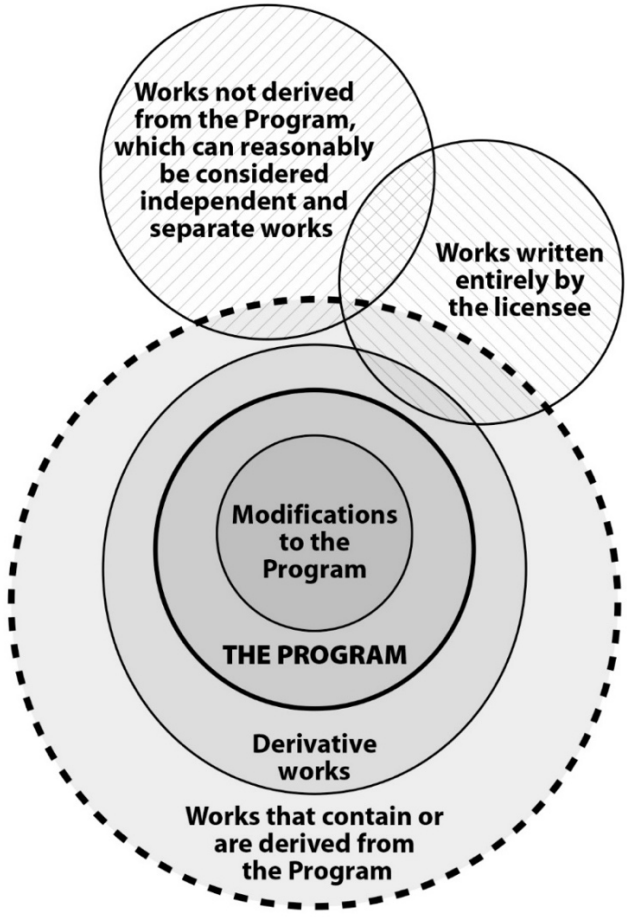


Figure 8.1 Venn diagram of code covered and not covered by the GPL

In a properly drafted document, the area within the heavy dashed line, which represents code covered by the GPL, would not overlap with the crosshatched portions, which represent code that is not covered by the GPL.

The ambiguity of these terms is undeniable. The FSF itself, author and steward of the document, has published extensive FAQs on GPL 2. Now that we know that the four corners rule will not give us an answer,

we have to look elsewhere. One observation: if a lawyer ever offers to write you a long analysis of GPL's language on its face, it is not worth the fees you will have to pay. GPL needs to be interpreted in its technical and social context, the importance of which far eclipses the language on its face. In this respect, interpreting GPL is no ordinary legal task.

In this sense, the GPL is more like a statute than a contract—it's a single document applied to many persons and many software projects. The traditional approach to contract interpretation via course of performance evidence is seldom relevant for GPL. If FSF is viewed as a legislative body setting the rules for its community, then what it says about the contract may be paramount. But FSF is not a legislative body, whose powers would be limited by the Constitution or whose position of authority would depend on a political process like elections or appointments. It is a self-appointed group. Statutory enactments have the power to affect many people's lives, and this is why the government that makes them has limited powers. The FSF is, after all, only a not-for-profit corporation, which ultimately is not bound to answer to an electorate. Therefore, its statements about interpretation should not necessarily be given the weight of law—though we will find them useful and interesting, as described below.

Applying the Rules of Contract Construction to the Border Dispute

We now have several (not necessarily mutually exclusive) alternative methods to interpret GPL:

- Look to trade usage.
- Look to the statutory meaning of the term *derivative work* (discussed below).
- Use a "legal realism" approach—focus on the risk of enforcement instead of pure interpretation.

One reason the language of GPL may seem opaque is that it uses a term of art from US copyright law: a *derivative work*. Under US law, copyright covers *works of authorship*, which include a variety of works such as books, music, videos, and software. A *derivative work* is a

variation of the work that is close enough so that it is governed by the rights of the original author, but not close enough so as to be identical. Outside the software context, a derivative work might be a revision (such as a new edition of a book), a translation (such as a translation of a book into another language), or an incorporation of a work into a larger work in a creative way (such as into a medley that uses parts of other songs).

This is good for us in our task of interpreting GPL; it means we have a whole body of law to draw on for the meaning of this important phrase. The FSF has publicly declared that it considers the definition of work based on the Program to be identical to that of a derivative work under copyright law. Unfortunately, while the term *derivative work* is clearly ensconced in copyright law, the law says almost nothing about what a derivative work of software might be. Copyright law is much more robust on works like books, music, and films. After all, software is relatively new compared with copyright law, which has been around for centuries—copyright is even mentioned in the US Constitution as one of the powers of the federal government.[4] On the other hand, software has only been around since the 1940s, and it was not clearly considered a copyrightable work of authorship until much later.

The "Derivative Works" Question

To be precise, it is not accurate to call the border dispute a question of what is a derivative work. If you look at the case law, you will find many cases that discuss how much variation in the original work is required to make a derivative work. This is not the question we are asking at all. We are asking what constitutes an *infringing work* versus a separate, non-infringing work. However, the question is usually cast as whether the Application and Library are a derivative work of the Application or a collective work.

[4] "The Congress shall have Power ...[t]o promote the Progress of Science and useful Arts, by securing for limited Times to Authors and Inventors the exclusive Right to their respective Writings and Discoveries," United States Constitution, Article I, Section 8.

The Copyright Act says this:

> A "derivative work" is a work based upon one or more preexisting works, such as a translation, musical arrangement, dramatization, fictionalization, motion picture version, sound recording, art reproduction, abridgment, condensation, or any other form in which a work may be recast, transformed, or adapted. A work consisting of editorial revisions, annotations, elaborations, or other modifications which, as a whole, represent an original work of authorship, is a "derivative work."

The Copyright Act also defines a *collective work* as "a work ... in which a number of contributions, constituting separate and independent works in themselves, are assembled into a collective whole."

Copyright law allows a copyright owner to exclude others from making derivative works; however, it does not allow the owner to exclude others from creating collective works.

In the language of GPL, some things are derivative works (i.e., they require one to exercise copyright in the original work), and some are *mere aggregation* (i.e., a compilation of separate works). The difference between these two is the crux of the border dispute. These two possibilities map to the difference between a single derivative work and a collective work.

Before we delve into this, we can isolate one easy case. Most modern programs are delivered in *packages*. These packages can have different names. In Java programming, they are called JAR files. In Linux system programming, they are often called tarballs. In contemporary computing, they may be called containers. They are also sometimes called images—particularly when an entire system is delivered in binary form. Putting software in the same JAR, tarball, container, or image as GPL code does not mean that software has to be licensed under GPL. In other words, if we deliver our Application and the time function Library in the same tarball, that doesn't mean—by itself—that the Library must be governed by GPL. That question depends on more detail about how the two elements of software interact when they are running. The packaging only governs how they are delivered.

Returning again to our "time" question, do you know which case is right—one or two? Of course not. To answer this question, we need to determine which work is at issue: Is it the Application and the Library as a whole, or is it only one of the components?

US copyright law offers little guidance on the difference between a single work and a collective work. There is an area of the law that is instructive, however—the law of statutory damages. Under US copyright law, an author can choose to sue for actual damages (the actual economic harm to the copyright holder) or statutory damages. The law sets a maximum amount of statutory damages for each work of authorship. "[W]here separate copyrights have no separate economic value, whatever their artistic value, they must be considered part of [a] ... work for purposes of the copyright statute."[5]

The economic value test would yield different results in different cases. In the cases of the Application and the Library, it seems fairly clear that they have separate markets. Obviously, the Library has uses for many Applications. The Application may not run without the Library, but it might also be useful on other operating systems that use other similar time libraries. So they do seem to be separate works. But the rule we have extracted from statutory damages law is hardly clear-cut, so we still need to consider how to analyze our question if we have no clear answer on this point.

Essentially, there are three possibilities:

1. The Application and the Library are one work.
2. The Application and the Library are separate works, and taken together, they form a collective work.
3. The Application and the Library are separate works, but when they are run together, they form a single work.

If we assume the Application and the Library are one work, our analysis is done. The Library is the Program, and the new work consisting of the Library and the Application is a work based on the

[5] *RSO Records, Inc. v. Peri*, 596 F.Supp. 849, 862 n. 16 (S.D.N.Y.1984).

Program. But what if they are separate works? This is a much more complicated question.

The Curious Case of the Software Under Copyright

Let's start by assuming that the Application and the Library are two separate copyrightable works and that we, as the Application developer, plan to distribute only our Application and not the Library. (Assume that the Library will be publicly available, so we can reasonably expect our customers to already have it.) We would then ask whether the Application is substantially similar to the Library. What do they have in common? They have a programmatic interface in common—an API.[6] Can this make the Application a derivative work of the Library?

Copyright protects artistic expression. For a book, a movie, or a song, it is easy to understand what that means, but for a computer program, it is not so easy. Copyright does not protect functional elements. This principle is called the *idea/expression dichotomy* or the *merger doctrine*. Source code is not as flexible as natural language, and there are fewer creative ways to use it. For example, in most programming languages, there is only one way to put a value into a variable (such as `a=1;`); therefore, this element cannot be copyrightable. The more difficult question is how complicated the code must become to enjoy copyright protection. For example, say there are many such statements:

```
a=1;
b=1;
c=1;
```

Is this copyrightable? Probably not. But the more we add, the more likely there are many ways to write the code, some of which may be more

[6] If you are unfamiliar with APIs, header files, and the concept of linking, please read Chapter 2: A Tutorial on Computer Software.

elegant or expressive than others. Many languages will allow you to write the same code with different white space, such as:

```
a=1; b=1; c=1;
```

And in other languages you may be able to write:

```
a=b=c=1;
```

And as a functionally equivalent alternative, you could always write:

```
c=1;
b=1;
a=1;
```

or:

```
a=6/6;
b=.5*2;
c=SQRT(1); [using the square root function]
```

All these look a bit different, but is the choice to write these, rather than the first example, expressive or merely trivial? Conceptually, the problem with analyzing software copyright is that each statement like a=1 is probably not protectable on its own, but at a certain point all of the unprotectable elements put together become protectable. It is not a bright line at all.

We must add to this the idea that much of software is dictated either by efficiency or the constraints of the programming language. Anything that is dictated by language syntax rules, technical requirements (such as hardware requirements), or functional needs is not copyrightable. If you put all this together, any work of software is a patchwork of copyrightable and uncopyrightable elements, yet the whole may be protectable. It is like music—each note is not protectable, but a phrase or line of music probably is protectable. For software, though, there are many discontinuities in protection, so the law has special rules for analyzing the protectability of software.

What Does US Law Say?

To find out what US law has to say about our question, we look to the language of the Copyright Act and cases that have been decided by judges based on that Act. In our system of law—called a *common law* system—we have to consider both the statute and the cases that comment on it. The Copyright Act says, "In no case does copyright protection for an original work of authorship extend to any idea, procedure, process, system, method of operation, concept, principle, or discovery, regardless of the form in which it is described, explained, illustrated, or embodied in such work" (17 USC 102(b)). The notion embodied by this statute is sometimes referred to as the doctrine of *merger* or the *idea/expression dichotomy*. On its face, this spells grave news for software, which clearly is at its core a procedure, process, and method of operation. However, software is squarely covered by the copyright law as a "literary work."

One of the earliest fair use cases stated, "There are, and can be, few, if any, things which, in an abstract sense, are strictly new and original throughout. Every book in literature, science and art, borrows and must necessarily borrow, and use much which was well known and used before"[7] . This applies to software more than to any other kind of copyrightable work. Therefore, to build a framework for analyzing software copyright, some courts have adopted the "abstraction, filtration, and comparison" (AFC) test. This test determines whether one work accused of infringement actually does embody the copyright of another work. If it does, it is infringing (or derivative); if not, it is a separate work. Under the AFC test, the court goes through the following process:

- Abstract all expressive elements of the programs from their ideas.
- Filter out all unprotectable elements.
- Compare any remaining elements of creative expression.

[7] *Emerson v. Davies*, 8 F.Cas. 615, 619 (D. Mass. 1845)

If the remaining elements are substantially similar, then the accused work is infringing.

Unprotectable elements could include high-level program architecture, algorithms, or data structures. These elements are filtered out of both the original work and the accused work, and the remaining expressive elements remain. Then the original work and the accused work are compared. If they are substantially similar, then the accusing work is a derivative work of the original and therefore infringes its copyright.

The Ninth Circuit (which is the US appeals court that has jurisdiction over California and therefore much of the software industry) uses the Analytic Dissection test. This test is similar, but not identical, to AFC. The test distinguishes between *intrinsic* and *extrinsic* components. *Intrinsic* or *expressive* elements are assessed "from the standpoint of the ordinary reasonable observer, with no expert assistance." The steps used in this assessment are as follows:

- Identify the source of the alleged similarity between the works (similar to the comparison step of the AFC test).
- Determine whether any of the allegedly similar features are protected by copyright (similar to the filtration step in the AFC test).
- Decide whether similar and protectable elements enjoy broad or thin protection (facts and ideas get thin protection).
- Decide whether the works are substantially similar as a whole.

Almost all the cases on this test do not involve computer software and are hard to apply to computer software. The analytic dissection test comes from *Sid & Marty Krofft Television Productions, Inc. v. McDonald's Corp.* and was later used in *Apple Computer, Inc. v. Microsoft Corp*. The *Apple* case concerned infringement by the Microsoft Windows GUI of the Apple Macintosh desktop GUI. *Sid & Marty Krofft* involved the infringement by the MacDonald's Hamburglar character of the characters in the television show *H.R. Pufnstuf*. For these, the viewpoint is that of an "ordinary reasonable observer, with no

expert assistance." For software, of course, this viewpoint is just not useful. The ordinary reasonable observer has no idea what an API is at all. Also, it is sensible that an ordinary, reasonable observer would be able to make a reasoned judgment about whether puppets or GUIs look the same. For software, the question is not how the code looks but how expressive it is. The average person could not tell an expressive line of code from an unexpressive one.

The question of whether a programmatic interface, standing alone, is copyrightable is an unsettled matter of law. As this book goes to press, the *Oracle America v. Google* case, centering on this issue, is still in the appeals process. Resolution of that case may clarify this issue, but until then, the case law on the question is laid out below.

In the *Worlds of Wonder* cases, the plaintiff sold a talking, dancing bear named Teddy Ruxpin. The plaintiff held a copyright on the toy. The programming for the bear's movement and voice was stored on cassette tapes built into the toy. The defendants had changed the tapes, thereby changing the programming. The Ninth Circuit held that third-party tapes were infringing derivative works, even though the third-party tapes did not contain any part of the recordings from the original tapes. However, the copyright in this case subsisted in the toy as an audiovisual work. Software code is a literary work. It is easy to see why the test for similarity for audiovisual works would be different and why the entire work could be infringing even though the cassettes were entirely different. The test for substantial similarity of an audiovisual work is to compare the "total concept" of the accused work with that of the original. Moreover, the court observed that children were the primary audience for the work and based its analysis on a child's point of view.

Micro Star v. Formgen, or the Duke Nukem case, involved the popular first-person shooter video game *Duke Nukem 3D*. The distributed game included a build editor that allowed users to create their own levels. Micro Star created a CD-ROM from 300 user-created levels and distributed them. These levels did not incorporate any code or artwork from the game, but they used MAP files created by the build editor to instantiate levels using art libraries from the game. The court

said, "A copyright owner holds the right to create sequels ... and the stories told in the N/I MAP files are surely sequels, telling new (though somewhat repetitive) tales of Duke's fabulous adventures. A book about Duke Nukem would infringe for the same reason, even if it contained no pictures." The court held the work to be infringing. Even though this case involved software, the analysis in the opinion compared the audiovisual displays of the game, rather than the software code. Again, the focus was on an audiovisual work, not a literary work.

These cases hold that a work can be infringing even if it contains no part of the original, despite the clearly enunciated rule that an infringing work "must substantially incorporate protected material from the preexisting work."[8]

If all this seems very complicated and unlikely to answer our question, it might be a relief to look at the case law in the First Circuit. The seminal case on copyright in computer software is *Lotus v. Borland*.[9] In this case, the accused work was a computer menu command hierarchy. The court said, "When faced with nonliteral-copying cases, courts must determine whether similarities are due merely to the fact that the two works share the same underlying idea or whether they instead indicate that the second author copied the first author's expression." The court found the command menu hierarchy to be an uncopyrightable "method" as described in 17 USC 102(b):

> The Lotus menu command hierarchy ... serves as the method by which the program is operated and controlled. ... The Lotus menu command hierarchy is also different from the Lotus screen displays, for users need not "use" any expressive aspects of the screen displays in order to operate Lotus 1-2-3; because the way the screens look has little bearing on how users control the program, the screen displays are not part of Lotus 1-2-3's "method of operation." The Lotus menu command hierarchy is also different from the underlying computer code, because while code is necessary for the program to work, its precise formulation is not. In other words, to offer the same capabilities as Lotus 1-2-3, Borland did not have to copy Lotus's underlying code (and indeed it did not); to allow users to

[8] *Micro Star v. FormGen Inc.* 154 F.3d 1107 (9th Cir. 1998).

[9] 49 F.3d 807 (1995).

operate its programs in substantially the same way, however, Borland had to copy the Lotus menu command hierarchy.

Lotus v. Borland remains the most instructive case for analyzing questions of protectability of APIs. The menu structure was, in fact, a kind of API—though between the macro writer and the spreadsheet program instead of between two programs.

Stepping a bit further afield into analogies, the LEXIS/Westlaw cases also analyze the bleeding edge of copyright. LEXIS and Westlaw are two leading companies in the field of reference books containing the published opinions of law courts. The texts of the opinions are not protected by copyright because they are works produced by the US government and therefore in the public domain. When Westlaw sued the owners of LEXIS for copying its material, it based the claim on the pagination of the cases in Westlaw's books. Mead had added a feature to its online products enabling the user to skip to a particular page in the opinion, as it would appear in the West reporter. In an initial decision in 1986, the Eighth Circuit found the use to be infringing. However, in light of the 1990 Supreme Court case *Feist Publications, Inc. v. Rural Telephone Service Co.*, which rejected the "sweat of the brow" doctrine in assessing copyright infringement, the same issue was later otherwise decided by the Second Circuit. As a consequence, use of West's paging is probably not considered infringement.

Even if a computer software interface is theoretically protectable under copyright, the reuse of small amounts of code for purposes of interoperability is often fair use. *Sega v. Accolade*[10] held that copying an initialization code in software for Sega Genesis game cartridges was fair use. *Vault Corp. v. Quaid Software, Ltd.* held that copying and decompilation of a software program key and fingerprint was fair use. APIs are close to initialization codes and program keys in this respect—they are needed for interoperability. In fact, APIs are the very definition of interoperability.

[10] 977 F.2d 1510 (9th Cir. 1992).

Summary

Overall, the law points to the notion that programmatic interfaces are likely to be functional and therefore not copyrightable. This, in turn, would point to the conclusion that the Application does not contain any protectable part of the Library, that it is therefore not a derivative work of the Library, and that the Application could therefore be distributed separately under a different license without violating GPL. But the law is very unsettled, particularly due to the outstanding appeal in *Oracle America v. Google*.

In sum, copyright law suggests that the Application and the Library are separate works, that neither is a derivative work of the other, and that the two taken together—regardless of whether they are integrated by dynamic linking, static linking, or otherwise—are a collective work rather than a single derivative work. However, as we will see below, the answer we get from a pure copyright analysis may not work in the real world of risk assessment.

International Interpretations

The above derivative-works analysis presumes the application of US law. However, because the GPL has no choice-of-law provision and relies for its power on background copyright law, the scope of derivative works can be different in different jurisdictions. A thorough analysis of this question is beyond the scope of this book. However, the answer to the border dispute might differ across jurisdictions, depending on how much weight a local jurisdiction gives to the protectability of APIs.

The Approach of Legal Realism

The above analysis is important to understand—if only to know how complicated the legal questions surrounding GPL might be in an actual lawsuit. But businesses, faced with making decisions about how to comply with GPL every day in a murky legal landscape, don't usually need to engage in this kind of legal analysis. They make decisions based on their estimate of real-world risk. We now leave behind the pure legal

analysis and examine best practices in a world where we assume the answer to the border dispute is unclear and is not likely to be cleared up anytime soon.

Quite a bit has been written on the Web about the border dispute, but much of it is confusing, uninformed, or mere rhetoric without logic. Keep in mind that the most important opinion about the application of GPL to any code belongs to the owner of the copyright in the code. If the copyright owner says that a particular practice—like linking the code to proprietary applications or using it as a library with proprietary code—is acceptable, then both the legal question and the risk question are resolved. But we might also consider the opinions of others, primarily because they reflect industry practice and community norms. Those offering the most frequent or useful opinions on this topic are the FSF. Recall that under a pure legal analysis, the statements we read by thought leaders the in the open source world are only of limited value as evidence of trade usage. But these statements matter a great deal to assessing real risk.

The FSF takes a functional approach to this question. Though that approach may seem arbitrary, at least it is useful. You may have heard that linking is the crux of this issue. That is not exactly true, but understanding linking is important to assessing the issue. (If you don't know what linking is, please take a few minutes to read Chapter 2: A Tutorial on Computer Software.)

Under this functional approach, the Application and the Library would either be considered part of the same Program or not, depending on how they interact but not on how they are packaged—including whether the Application developer is actually distributing the Library or not.

Even if the two are delivered separately, if they are considered part of the same work, the Free Software Foundation expressly states they

must both be under GPL. The FSF has released a FAQ on the interpretation of GPL 2, and that FAQ says this:[11]

> Q: You have a GPL'ed program that I'd like to link with my code to build a proprietary program. Does the fact that I link with your program mean I have to GPL my program?
>
> A: Not exactly. It means you must release your program under a license compatible with the GPL (more precisely, compatible with one or more GPL versions accepted by all the rest of the code in the combination that you link). The combination itself is then available under those GPL versions.

But the FSF also says:

> Q: If I write a plug-in to use with a GPL-covered program, what requirements does that impose on the licenses I can use for distributing my plug-in?
>
> A: It depends on how the program invokes its plug-ins. If the program uses fork and exec to invoke plug-ins, then the plug-ins are separate programs, so the license for the main program makes no requirements for them.
>
> If the program dynamically links plug-ins, and they make function calls to each other and share data structures, we believe they form a single program, which must be treated as an extension of both the main program and the plug-ins. This means you must license the plug-in under the GPL or a GPL-compatible free software license and distribute it with source code in a GPL-compliant way.
>
> If the program dynamically links plug-ins, but the communication between them is limited to invoking the "main" function of the plug-in with some options and waiting for it to return, that is a borderline case.[12]

Many people find this discussion of linking confusing. While some programmers may find it less so, it is still misleading. First, linking is a concept that only exists for certain kinds of programming languages, like C++. When you are programming in languages like PERL, PHP, or HTML (so-called *scripting languages*), the concept of linking is

[11] www.gnu.org/licenses/gpl-faq.html#LinkingWithGPL. This link is historical and has been superseded on the current FAQ.

[12] www.gnu.org/licenses/gpl-faq.html#GPLAndPlugins.

meaningless or nearly so. The bottom line here is that for GPL interpretation, the method of linking is a distinction without a difference. This makes sense because in a program build, it is usually possible to either dynamically or statically link any code of your choosing, depending on how you think the program will best perform. Static linking makes the program larger but eliminates processing time used to find the linked code. Dynamic linking makes more efficient use of memory while slowing down processing speed.

At one time, there was a notion floated in the open source world that dynamic linking mattered. In other words, in our example, if the Application and the Library were dynamically linked, they were two different programs, and if they were statically linked, they were the same program. Clearly, if they are statically linked, they are in the same binary file, but not if they are dynamically linked. But in practice, this is not a useful distinction.

Eric Raymond—one of the pioneers of open source and one of the founders of the Open Source Initiative—discussed this in a Web exchange regarding Linus Torvalds's views on whether loadable kernel modules or LKMs create derivative works of the Linux kernel under the GPL. Someone commented that "Linus's opinion on this is irrelevant," and Raymond said:

> [I]n fact, I agree with his assessment. The key question is whether the particular kind of linking involved with loading binary modules propagates derivative-work status under copyright law. This is a legal question a court may rule on someday. Until one does, anyone who relies on such linking is taking a legal risk. ... [But it] is not quite right that Linus's opinion is irrelevant. It is irrelevant to the underlying legal question, but not to the associated business risk.

And this is exactly so.

The FSF's View

The FSF has made the most, and the most thoughtful, public commentary on the border dispute. Therefore, we take their position as a baseline best practice. Their position is as follows:

- Any linking (dynamic or static) to GPL code creates a single, derivative work of that code that falls within the boundary.
- Software that interacts via communications protocols such as pipes and sockets is not a derivative work.
- Software programs that interact only via shell commands and exec statements are separate works.
- User space is outside the boundary of the GPL applied to the kernel.
- Source code for a GPL program does not include standard Linux system or language libraries.

The FAQ on the GPL 2 says this:[13]

> Q: Where's the line between two separate programs, and one program with two parts?
>
> A: This is a legal question, which ultimately judges will decide. We believe that a proper criterion depends both on the mechanism of communication (exec, pipes, rpc, function calls within a shared address space, etc.) and the semantics of the communication (what kinds of information are interchanged).
>
> If the modules are included in the same executable file, they are definitely combined in one program. If modules are designed to run linked together in a shared address space, that almost surely means combining them into one program.
>
> By contrast, pipes, sockets, and command-line arguments are communication mechanisms normally used between two separate programs. So when they are used for communication, the modules normally are separate programs. But if the semantics of the communication are intimate enough, exchanging complex internal data structures, that too could be a basis to consider the two parts as combined into a larger program.[14]

According to FSF, therefore, anything that is linked in the same executable as GPL code must be covered by GPL. If our Application developer adheres to this rule, he will be safe. However, there are

[13] This language has been updated to a new question on https://www.gnu.org/licenses/gpl-faq.en.html#MereAggregation but the substance is the same.

[14] gnu.org/licenses/gpl-faq.html#MereAggregation.

exceptions to this rule, and the exceptions can be challenging to understand, particularly for non-engineers.

But in fact, the above rule does not fully enunciate the FSF's view—it is merely an expedient to avoid having one's coding practices fall outside FSF's approval. The FSF often refers to how "intimately" programs are integrated, and clearly this notion is an ad hoc or fluid one.

Because, in this sense, there is never a clear answer to the border dispute question, it is a good idea to follow the spirit of GPL, even when you can't tell whether you are following its letter. The spirit of the GPL is freedom—anyone who gets binaries should be able to get source code. Anyone who uses a binary should be able to fix it, study it, and change it. If you are pondering the border disputes because you are trying to hide functionality in proprietary modules, that is not the spirit of GPL. However, if the interface between proprietary modules and GPL modules is transparent, simple, well documented, and a true black box, then even if you have violated GPL, fewer people are likely to complain about it. And that, in a nutshell, is risk management. Moreover, simple, clear interfaces are good engineering. So the best approach to comply with GPL is to do the right thing in engineering your software.

Loadable Kernel Modules (LKMs)

The border dispute of GPL crystallized in the controversy over loadable kernel modules. For the most part, the Linux kernel is *monolithic*, meaning it is a single binary that loads when the computer boots up. But it can also support dynamically loaded modules. The poster child for this debate is a company called Nvidia, which has proprietary Linux drivers for its graphics card drivers—causing Richard Stallman to nickname the company "Invidious."[15]

However, other free software advocates and other Linux developers—such as Linus Torvalds—have expressed other views on the

[15] A caveat: It's hard to find evidence of this epithet on the Web. Some links on the topic appear to be broken.

border dispute, particularly as it relates to LKMs. Torvalds's comments are particularly interesting:

> There are (mainly historical) examples of UNIX device drivers and some UNIX filesystems that were preexisting pieces of work, and which had fairly well-defined and clear interfaces and that I personally could not really consider any kind of "derived work" at all, and that were thus acceptable. The clearest example of this is probably the AFS (the Andrew Filesystem), but there have been various device drivers ported from SCO too.[16]

Torvalds is, of course, taking a fact-specific and practical view of the question. Linux was originally written to implement the interface specification of UNIX, so Torvalds naturally had assumed that the interface was free to use. He has also said this about the exception for system libraries:

> Well, there really is no exception. However, copyright law obviously hinges on the definition of "derived work," and as such anything can always be argued on that point.
>
> I personally consider anything a "derived work" that needs special hooks in the kernel to function with Linux (i.e., it is _not_ acceptable to make a small piece of GPL-code as a hook for the larger piece), as that obviously implies that the bigger module needs "help" from the main kernel.
>
> Similarly, I consider anything that has intimate knowledge about kernel internals to be a derived work.
>
> What is left in the gray area tends to be clearly separate modules: code that had a life outside Linux from the beginning, and that do[es] something self-contained that doesn't really have any impact on the rest of the kernel. A device driver that was originally written for something else, and that doesn't need any but the standard UNIX read/write kind of interfaces, for example.

He has also said this:

> Well, see above about the lack of exception, and about the fundamental gray area in _any_ copyright issue. The "derived work" issue is obviously a gray area, and I know lawyers don't like them. Crazy people (even

[16] Quotations from Torvalds in this section are all from www.win.tue.nl/~aeb/linux/lk/COPYING-modules.txt.

> judges) have, as we know, claimed that even obvious spoofs of a work that contain nothing of the original work itself, can be ruled to be "derived."
>
> I don't hold views that extreme, but at the same time I do consider a module written for Linux and using kernel infrastructures to get its work done, even if not actually copying any existing Linux code, to be a derived work by default. You'd have to have a strong case to _not_ consider your code a derived work.

He has also observed that the more complex Linux gets, the less likely it is that an LKM should be considered a separate work:

> The Linux kernel modules had (a long time ago), a more limited interface, and not very many functions were actually exported. So five or six years ago, we could believably claim that "if you only use these N interfaces that are exported from the standard kernel, you've kind of implicitly proven that you do not need the kernel infrastructure."
>
> That was never really documented either (more of a guideline for me and others when we looked at the "derived work" issue), and as modules were more and more used not for external stuff, but just for dynamic loading of standard Linux modules that were distributed as part of the kernel anyway, the "limited interfaces" argument is no longer a very good guideline for "derived work."
>
> So these days, we export many internal interfaces, not because we don't think that they would "taint" the linker, but simply because it's useful to do dynamic run-time loading of modules even with standard kernel modules that _are_ supposed to know a lot about kernel internals, and are obviously "derived works."

Torvalds takes the practical view that published interfaces support the existence of a separate work.

> It's an issue of what a "plug-in" is—is it a way for the program to internally load more modules as it needs them, or is it _meant_ to be a public, published interface.
>
> For example, the "system call" interface could be considered a "plug-in interface," and running a user mode program under Linux could easily be construed as running a "plug-in" for the Linux kernel. No?
>
> And there, I obviously absolutely agree with you 100%: the interface is published, and it's _meant_ for external and independent users. It's an interface that we go to great lengths to preserve as well as we can, and it's an interface that is designed to be independent of kernel versions.

> But maybe somebody wrote his program with the intention to dynamically load "actors" as they were needed, as a way to maintain a good modularity, and to try to keep the problem spaces well defined. In that case, the "plug-in" may technically follow all the same rules as the system call interface, even though the author doesn't intend it that way.
>
> So I think it's to a large degree a matter of intent, but it could arguably also be considered a matter of stability and documentation (i.e., "require recompilation of the plug-in between version changes" would tend to imply that it's an internal interface, while "documented binary compatibility across many releases" implies a more stable external interface, and less of a derived work).

All of this underscores that there is no clear, identifiable difference between the way a program operates through a "standard" interface and any other kind of interface. This is part of the reason the border dispute remains so intractable—what constitutes a platform interface, a language interface, or a "standard" interface is fact specific and sometimes, ultimately, a matter of industry consensus. Moreover, it changes over time, as computing becomes increasingly layered.

Export Symbol

The "Export Symbol" method of determining GPL compliance is less popular than it once was, but it remains an interesting window into the expression of an author's intent on difficult license interpretation questions, particularly for a complex project like the Linux kernel. The kernel contains a "MODULE_LICENSE" macro. An author of a module of Linux can include code so that invoking the macro will return a certain value if queried—it is a way to "hard code" licensing information into the software. For instance, if a module contains the statement MODULE_LICENSE("GPL"), it means the author intends that the license for any module being coded to that interface should be GPL. In other words, this is a way for the author to express his intent that the interface is part of the "Program" covered by GPL. Historically, Linux programmers have often followed these signals—LKMs coded to such an interface would be licensed under GPL. Compile/build systems can be set to generate errors if the wrong kind of licensing information applies to a module being used.

Here is an interesting discussion entitled "On the value of EXPORT_SYMBOL_GPL":[17]

> When a loadable module is inserted, any references it makes to kernel functions and data structures must be linked to the current running kernel. The module loader does not provide access to all kernel symbols, however; only those which have been explicitly exported are available. The export requirement narrows the API seen by modules, though not by all that much: there are over 6,000 symbols exported in the 2.6.13 kernel.
>
> Exports come in two flavors: vanilla (EXPORT_SYMBOL) and GPL-only (EXPORT_SYMBOL_GPL). The former are available to any kernel module, while the latter cannot be used by any modules which do not carry a GPL-compatible license. The module loader will enforce this distinction by denying access to GPL-only symbols if the module's declared license does not pass muster. Currently, less than 10% of the kernel's symbols are GPL-only, but the number of GPL-only symbols is growing. There is a certain amount of pressure to make new exports GPL-only in many cases.

It is a more difficult question, though, as to whether the absence of a GPL-only symbol means the interface could be used by a proprietary LKM. Silence may not be consent. In legal terms, an express statement that linking to proprietary code is allowed could be viewed as waiver or estoppel. However, that doesn't necessarily mean the converse: that failing to object to such integration means it is allowed. Moreover, with each new release of Linux, more modules are coded as GPL-only. Therefore, relying on EXPORT_SYMBOL analysis to analyze the border dispute is not a stable model and should be approached with caution.

The Other Shoe Drops, but Drops Away

Many in the open source legal community have been awaiting the day when the GPL 2 border dispute will be clarified by judicial opinion. A recent case promised to shed light on this question. In 2015, a Linux kernel contributor brought suit in Germany against VMware, the most

[17] http://lwn.net/Articles/154602/.

successful maker of virtualization software. The case involved, in a way, the opposite of the quintessential boundary question, which is whether it is compliant to add proprietary drivers to the Linux kernel. This case addressed whether Linux drivers could be added to the proprietary VMware kernel. However, the case was dismissed in August 2016 on procedural grounds before the court could come close to a substantive decision. For more information, see Chapter 19: Enforcement and Obstacles to Enforcement.

Chapter 9

LGPL 2.1 Compliance

LGPL version 2.1 may be one of the most abstruse licenses ever written. While its best practices for compliance are less difficult to understand than those of GPL, mapping those practices onto the text of LGPL is difficult.

The Lesser General Public License was written to enable authors to release libraries under a GPL-style license. Because GPL would require all code integrated into the same program as the library to be licensed under GPL, GPL does not work for libraries intended for use in proprietary applications. LGPL, therefore, was intended to relax some of the requirements of GPL to allow the license to be applied to libraries. LGPL 2.1 is an entirely separate license, so it is not obvious on its face that LGPL 2.1 is intended to be a variation of GPL.[1]

The basic rule of LGPL compliance (as understood by everyone) is that in proprietary applications, LGPL should be used only for dynamically linked libraries. But that rule is a best practice more than a requirement. The core terms in Section 6 of LGPL 2.1 are as follows:

> As an exception to the Sections above, you may also combine or link a "work that uses the Library" with the Library to produce a work containing portions of the Library, and distribute that work under terms of your choice, provided that the terms permit modification of the work for the customer's own use and reverse engineering for debugging such modifications. ... Also, you must do one of these things:
>
> a) Accompany the work with the complete corresponding machine-readable source code for the Library including whatever changes were used in the work ... and, if the work is an executable linked with the

[1] In the version 3 licenses, this was done in a more understandable way—LGPL 3 is a set of additional permissions that apply as a supplement to the terms of GPL 3.

> Library, with the complete machine-readable "work that uses the Library," as object code and/or source code, so that the user can modify the Library and then relink to produce a modified executable containing the modified Library.
>
> b) Use a suitable shared library mechanism for linking with the Library. A suitable mechanism is one that (1) uses at run time a copy of the library already present on the user's computer system, rather than copying library functions into the executable, and (2) will operate properly with a modified version of the library, if the user installs one, as long as the modified version is interface-compatible with the version that the work was made with.

The well-known compliance rule regarding dynamic linking comes from Section (b) above—dynamic linking is a shared library mechanism. Most companies, as a best practice, limit their use of LGPL code to dynamically linked libraries.

There is another way to comply, but most companies consider it riskier and less attractive. You can, for example, statically link LGPL code to your application if you both

- provide the source code for the Library; and
- provide the complete machine-readable "work that uses the Library," as *object code and/or source code*.

This requires delivery of unlinked objects for proprietary software. While this would not normally require disclosure of proprietary code or information, it is an administrative burden to create such a package to deliver—a package unlikely to be used in precisely that form, even by the code developer. Also, lawyers are troubled by the "and/or" because they are concerned it will be interpreted as "and" and require delivery of source code for the application. It's likely that the better interpretation is that it does not require delivery and is thus intended to be a Boolean "or." But this ambiguity of "and/or"[2] is a drafting lesson that lawyers learn the hard way. In any case, the application developer usually does

[2] I am compelled to comment here that, like many lawyers, I hate the slash construction and never use it. I would rather include interpretive guidance that "or" is considered inclusive rather than exclusive (the exclusive form being, in programming, XOR), if the distinction must be made.

not wish to make the proprietary objects available for relinking or doesn't have the rights to do that due to upstream licensing restrictions for third-party proprietary objects.

Other complexities of LGPL compliance include the question of reverse engineering. LGPL 2.1 contains a provision that can conflict with most end user licenses. Section 6 says this:

> [Y]ou may also combine or link a "work that uses the Library" with the Library to produce a work containing portions of the Library, and distribute that work under terms of your choice, provided that the terms permit modification of the work for the customer's own use and reverse engineering for debugging such modifications."

Most proprietary licenses restrict reverse engineering, and they therefore conflict with the provision of LGPL 2.1 that requires the entire application to be provided on terms that "permit modification of the work for the customer's own use and reverse engineering for debugging such modifications." If a proprietary application is properly integrated with the LGPL code, then the interface is a black box and no such reverse engineering should be necessary. Therefore, this may not be a significant substantive concern for developers of proprietary applications, but it is a trap for the unwary—it's easy to violate LGPL by forgetting to carve out reverse-engineering restrictions to the extent necessary to allow this.

To comply with LGPL, the end user license must be revised to make an exception for the LGPL code. Practically speaking, this is not difficult, because any end user license for a product that includes LGPL code must contain carve-outs; LGPL code cannot be sublicensed and must be licensed only on the terms allowed by LGPL. For a discussion of how to resolve the two, see Chapter 17: Mergers & Acquisitions and Other Transactions.

Anyone responsible for compliance with LGPL should also note that some technology platforms, such as iOS and Android, do not

support dynamic linking. Therefore, LGPL code is seldom appropriate for mobile applications.[3]

Finally, there is the 10-line limit on source code macros. LGPL limits the use, via static linking or *include* statements, to 10 lines. (For more explanation of in-line functions, see Chapter 2: A Tutorial on Computer Software.) This restriction is very limiting. Contemporary compilation techniques not only make use of more static linking and in-line compilation (because more weight is given to processing speed than memory limitations), but the 10-line limit may be based on a urban legend that anything under 10 lines of software is not subject to copyright.[4] While this 10-line limitation is potentially a thorny issue, it is pretty much ignored in practice.

[3] This is a conservative rule. It is possible to comply with LGPL in ways other than using the code via dynamic linking. Moreover, some authors of libraries that are useful in mobile applications make public statements that they approve the use of the code in mobile applications via static linking.

[4] There is no such rule under law per se; however, 10 lines is often given as an example threshold below which use would constitute fair use (a defense to infringement), or the 10 lines would not be protectable (and therefore there would be no infringement).

Chapter 10

GPL 3 and Affero GPL 3

GPL version 2 was released in 1991. As the twenty-first century approached, it was clear that a revision was in order. In the interim, the license had gained impressive traction, the software industry had changed, and some laws affecting software licensing had changed as well. The revision process was long and involved many stakeholders. The revised licenses have been in existence for some years now. This chapter outlines the differences between the new, version 3 licenses and their predecessors, and it describes how the licenses apply to the current software landscape.

GPL Version 3.0 (GPL 3)

Version 3 of the GPL was released on June 29, 2007. Since that time, its adoption has been increasing slowly. Although many software projects have been released under GPL 3—or GPL 2 or any later version, which allows use under GPL 3—many projects remain under GPL 2 only.

The reluctance to follow FSF's lead in moving to the new version demonstrates that the revision happened at a time when the free software movement had lost traction to more practical voices, particularly those from private industry. Although private industry had, somewhat reluctantly, embraced free software, it had not embraced free software ideals or the leadership of the FSF. The GPL 3 revision project was long and difficult. During the process, there were acrimonious public arguments between key players—for instance, FSF and Linus Torvalds (and the kernel maintainers), who were unconvinced of the need for a new license. James Bottomley and others wrote:

> Since GPL has served us so well for so long, and since it is the foundation of our developer contract which has helped propel Linux to the successes it enjoys today, we are extremely reluctant to contemplate tampering with that license except as bug fixes to correct exposed problems or updates [to] counter imminent dangers. So far, in the whole history of GPLv2 ... we have not found any bugs serious enough to warrant such correction.[1]

These disputes were ostensibly resolved by the time the final draft was issued, but they signaled deep philosophical differences in the open source community.

Also, during the redrafting project, Microsoft and Novell announced a patent deal. The terms of this deal were disclosed on a confidential basis to SFLC,[2] and the terms were soon thereafter filed in connection with Novell's Annual Report on Form10-K. This delayed the final draft and resulted in the so-called "Anti-Microsoft" and "Anti-Novell" provisions described below.

There is no doubt that GPL 2 needed to be improved and updated, but several of the new features that were added to GPL 3 were met with lukewarm acceptance, particularly among commercial enterprises. On the other hand, GPL 3 added some useful and clarifying terms. A detailed explanation of many of the changes embodied in GPL 3 appears in the FSF's rationale document for the last discussion draft.[3]

Versioning of Licenses

Most copyleft licenses have versioning rules baked into them. Section 9 of GPL 2 said:

> The Free Software Foundation may publish revised and/or new versions of the General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. Each version is given a distinguishing

[1] James Bottomley et al., "GPLv3 Position Statement," September 22, 2006, http://lkml.org/lkml/2006/9/22/217.

[2] Tom Sanders, "Novell opens legal books to GPL pundits," November 9, 2006, http://itnews.com.au/news/novell-opens-legal-books-to-gpl-pundits-68071.

[3] Available at http://gplv3.fsf.org/gpl3-dd4-rationale.pdf.

> version number. If the Program specifies a version number of this License which applies to it and "any later version," you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of this License, you may choose any version ever published by the Free Software Foundation.

Most software that had been released under GPL prior to the publication of version 3 was licensed under "GPL version 2 or any later version." At that point, all such software was available under either version, at the recipient's choice. The Linux kernel, however, was mostly under version 2 only.

In 2001, Torvalds wrote:

> I don't trust the FSF. I like the GPL a lot—although not necessarily as a legal piece of paper, but more as an intent. Which explains why, if you've looked at the Linux COPYING file, you may have noticed the explicit comment about "only _this_ particular version of the GPL covers the kernel by default". That's because I agree with the GPL as-is, but I do not agree with the FSF on many other matters. ... The FSF has long been discussing and is drafting the "next generation" GPL, and they generally suggest that people using the GPL should say "v2 or at your choice any later version." ... The "v2 only" issue might change some day, but only after all documented copyright holders agree on it, and only after we've seen what the FSF suggests. From what I've seen so far from the FSF drafts, we're not likely to change our v2-only stance, but there might of course be legal reasons why we'd have to do something like it (i.e., somebody challenging the GPLv2 in court, and part of it to be found unenforceable or similar would obviously mean that we'd have to reconsider the license).[4]

As Torvalds accurately pointed out, those who choose to license their software—even optionally—under any later version are placing a great deal of trust in the license steward.

In fact, because the kernel development project was run without a contribution agreement, it would probably be impossible to reconsider the license.[5] GPL 2 and GPL 3 are not compatible—a program has to be

[4] http://lkml.org/lkml/2003/6/15/67.

[5] For more on why, see Chapter 16: Open Source Releases.

under either one or the other.[6] Therefore, it is possible to combine in one program code that is under GPL 3 and "GPL 2 or any later version"—this would mean the whole program would be licensed under GPL 3. However, it is not possible to combine software under GPL 3 and "GPL 2 only."

Assuming a project had been available under either version, the project could, at any point, have chosen only the later version. This "migration" was not undertaken by many projects after the release of GPL 3; it was undertaken by the projects run by FSF, such as GCC and other GNU tools.

GPL 2 Versus GPL 3

GPL 3 was an entire rewrite of the license; little of the original text remains. However, there are only a handful of significant substantive differences; most of the changes were intended to clarify the original intent of GPL 2 and update it to meet the legal and technical context of the twenty-first century. The substantive differences, roughly in order of importance, are as follows:

- Derivative works (or scope) definition
- Definition of copyleft trigger
- Patent licensing
- Other patent provisions
- Obfuscation and disabling ("anti-Tivoization")
- DRM provisions
- DMCA provisions
- Effect of corporate transactions

[6] Richard Stallman, "Why Upgrade to GPL Version 3?" a communication distributed to the comment committee distribution list on May 31, 2007.

The "Derivative Works" Issue

The biggest question for most business users of GPL software has to do with its scope: What software must be made available under GPL? GPL 2 caused great confusion on this issue.[7] GPL 3 took positive steps to clarify this question.

Whereas GPL 2 treated this subject primarily as a question of what is a derivative work, GPL 3 moved away from that calculus; the term *derivative work* is mainly drawn from US law, and one objective of GPL 3 was to internationalize the license.

GPL 3's formulation of scope turns mainly on the definition of *Corresponding Source*, which includes "definition files associated with source files for the work, and the source code for shared libraries and dynamically linked subprograms that the work is specifically designed to require, such as by intimate data communication or control flow between those subprograms and other parts of the work." Whereas GPL 2's inclusion of dynamically linked files within its scope was left to external interpretive evidence, GPL 3 incorporated some of the language of the FSF's prior FAQs on this issue for GPL 2. While "intimate" communication may still be subject to interpretation, GPL 3 has clarified that dynamically linked files are required.

GPL 3 also clarifies that standard language libraries need not be covered by GPL even if they link to GPL code. In the intervening time between GPL 2 and GPL 3, computing became more layered: today's computing environment usually includes multiple language platforms or virtual machines. Rather than drawing a single line between user space and operating system space, GPL 3 allows integration of multiple platform layers without requiring a single licensing paradigm to cover them all.

[7] For a detailed analysis, see Chapter 8: The GPL 2 Border Dispute.

Copyleft Triggers

The trigger for copyleft conditions for GPL 2 was distribution. (See Chapter 6: What Is Distribution? for more detail on this issue.) However, the term *distribution* is somewhat specific to US copyright law. Because GPL 3 was intended to internationalize and clarify the license, its language was greatly changed to avoid turning on a definition drawing from US law. During the drafting process, opinions differed greatly on whether to close the *cloud services loophole*—the ability to provide software on a SaaS basis and not make source code available. In the years between the publication of GPL 2 in 1991 and the drafting project for GPL 3 in 2007, the software industry had undergone a sea change away from on-premises installation of software and toward SaaS. Some felt that this allowed free riders to make significant commercial use of GPL software without sharing source code; others thought that changing this rule would create havoc in an industry that had based compliance processes on whether software was distributed.

In GPL 3, two new terms of art were introduced: *conveying* and *propagating*. Only *conveying* triggers copyleft requirements. GPL 3 specifically states that "[m]ere interaction with a user through a computer network, with no transfer of a copy, is not conveying." *Propagating* is a broader term that might include what some refer to (confusingly and inaccurately) as *internal distribution*, or SaaS use.

Patents

GPL 2 contains no express patent licenses. Between the time GPL 2 was released in 1991 and the GPL 3 revision process in 2007, many other open source licenses were revised to contain express patent licenses. During that time, software patents became more common. FSF took the position that although there was no express patent license in GPL 2, there was an implied license. Because the law of implied licensing is unclear, the scope of that license was unclear.

The scope of the express license in GPL 3 was a compromise; the patent terms were lightning rods of controversy in the drafting

discussions. In the end, GPL 3 contained express licenses that were similar to, but probably slightly broader than, those of other copyleft licenses and Apache 2. Like the terms of the other licenses, the patent grant in GPL 3 only inured to contributors to the software. Thus, a "mere redistributor" granted no right under the express license. The license extended to a contributor's "essential patent claims," which are patents "owned or controlled" by the contributor "whether already acquired or hereafter acquired."

Most other open source licenses with express patent grants contain defensive termination provisions. GPL 3 does not, exactly, but contains terms with a similar effect in Section 10:

> You may not impose any further restrictions on the exercise of the rights granted or affirmed under this License. For example, you may not impose a license fee, royalty, or other charge for exercise of rights granted under this License, and you may not initiate litigation (including a cross-claim or counterclaim in a lawsuit) alleging that any patent claim is infringed by making, using, selling, offering for sale, or importing the Program or any portion of it.

Accordingly, any patent claim by a licensee accusing the GPL 3 software will be a breach of the license and terminate not only the patent license from others but the copyright license as well.

The patent license grant was not terribly controversial in the end. It was customary given similar terms in other open source licenses that had been in practice for years. The controversy centered on the patent provisions added in the final draft—the so-called "Anti-Microsoft" and "Anti-Novell" provisions. These provisions were added to the draft with little community input, near the end of the drafting effort, and they therefore did not benefit from much mindshare of the community on whether they made sense from either a substantive or a drafting perspective. Most readers of GPL 3 find them bewildering.

The Microsoft/Novell deal involved a promise to grant a patent non-assert by Microsoft for software distributed by Novell, for which Novell (and presumably its customers, ultimately) paid a fee. In this case, Microsoft avoided distributing GPL code—thereby avoiding a patent license being implied—but received a fee for granting the right

and also benefited from the related marketing deal with Novell. Apparently, FSF wished to make such arrangements inconsistent with the exercise of a license under GPL 3.

Section 11 of GPL 3 says this:

> If you convey a covered work, knowingly relying on a patent license, and the Corresponding Source of the work is not available for anyone to copy, free of charge and under the terms of this License, through a publicly available network server or other readily accessible means, then you must either (1) cause the Corresponding Source to be so available, or (2) arrange to deprive yourself of the benefit of the patent license for this particular work, or (3) arrange, in a manner consistent with the requirements of this License, to extend the patent license to downstream recipients. "Knowingly relying" means you have actual knowledge that, but for the patent license, your conveying the covered work in a country, or your recipient's use of the covered work in a country, would infringe one or more identifiable patents in that country that you have reason to believe are valid.

This paragraph is hard to understand at first blush but relatively easy to comply with by taking option (1) and making the Corresponding Source "available for anyone to copy, free of charge and under the terms of this License." Theoretically, causing the Corresponding Source to be available allows recipients of the software to more easily engineer around patent infringement claims. The provision's usefulness is questionable, but it does not present any thorny problems of interpretation. Anyone who conveys a work under GPL 3 has an obligation to make source code available; this provision only requires the source code to be available to any taker; the easiest route to compliance is to post the code on a publicly available website.

Another paragraph of Section 11 says this:

> If, pursuant to or in connection with a single transaction or arrangement, you convey, or propagate by procuring conveyance of, a covered work, and grant a patent license to some of the parties receiving the covered work authorizing them to use, propagate, modify or convey a specific copy of the covered work, then the patent license you grant is automatically extended to all recipients of the covered work and works based on it.

Of course, if the licensee ("you") does not have the right to grant a patent license to every recipient, this provision cannot make it so; presumably, the licensee that is unable to grant such broad rights will be in breach of the license. Therefore, a licensee ("you") who conveys a GPL 3 work must either clear patent rights for all recipients or for none. Of course, this excludes business models that might do otherwise, but few companies are interested in such strategies. This is probably the most troublesome provision of GPL 3. Many patent licenses are granted to settle end-patent litigation. A defendant (licensee) who receives a patent license in settlement will almost never be able to negotiate a broad enough right to comply with this provision. Therefore, faced with a third-party patent claim, the licensee may have to choose between settling the litigation and conveying the GPL 3 software. That is a business risk; it could cause the licensee to have to cease the use of the GPL 3 software, through no fault of its own.

Finally, Section 11 contains this language:

> A patent license is "discriminatory" if it does not include within the scope of its coverage, prohibits the exercise of, or is conditioned on the non-exercise of one or more of the rights that are specifically granted under this License. You may not convey a covered work if you are a party to an arrangement with a third party that is in the business of distributing software, under which you make payment to the third party based on the extent of your activity of conveying the work, and under which the third party grants, to any of the parties who would receive the covered work from you, a discriminatory patent license (a) in connection with copies of the covered work conveyed by you (or copies made from those copies), or (b) primarily for and in connection with specific products or compilations that contain the covered work, unless you entered into that arrangement, or that patent license was granted, prior to 28 March 2007.

This provision is clearly an attempt to prevent deals like the one Novell entered into with Microsoft; Novell had done roughly what was described here. However, the date at the end of the provision grandfathers in that deal. This was the date of the first discussion draft of GPL 3 that included language addressed at the deal, but the deal was entered into before that time.

DMCA

The Digital Millennium Copyright Act (DMCA) went into effect in 1998—between the publication of GPL 2 and GPL 3. Discussion of the DMCA can be confusing because the law covers several distinct subjects.[8] The element of concern for free software is the part that gets the most attention in the news: an odd addition to the copyright law that provides for civil and criminal penalties for the act of reverse engineering to circumvent copyright protection. The provision is odd because it covers actions not regulated by the main provisions of the copyright law. It was quickly invoked in the law to prevent actions that may never have been anticipated by its passage.[9] It was passed in part to enable the entertainment industry to implement digital rights management technology, or DRM.

Although the free software community tends to oppose DRM,[10] that is more an issue around content than software. However, a time-honored functionality of open source software is to hack all kinds of security measures. The anti-DRM provisions of GPL 3 went through quite a bit of revision in the drafting process and caused much disagreement among stakeholders. Some were concerned that any such provision would have to be so general that it would exclude legitimate security functionality. The Open Source Definition requires nondiscrimination as to all fields of endeavor, so it would not have been acceptable to exclude DRM—or any other kind of functionality—from being licensed under GPL 3, nor would it have been acceptable to exclude modifications to add that functionality. The final provision in GPL 3 is seldom invoked and was watered down to the point of non-controversy.

Section 3 of GPL 3 says this:

[8] The DMCA also provides a safe harbor for online service providers—a separate subject that sometimes gets tarred with the same brush.

[9] See *Chamberlain v. Skylink*, 381 F.3d 1178 (Fed. Cir. 2004); and *Lexmark Int'l, Inc. v. Static Control Components, Inc.*, 387 F.3d 522 (6th Cir. 2004).

[10] www.gnu.org/philosophy/opposing-drm.html.

Protecting Users' Legal Rights From Anti-Circumvention Law

> No covered work shall be deemed part of an effective technological measure under any applicable law fulfilling obligations under article 11 of the WIPO copyright treaty adopted on 20 December 1996, or similar laws prohibiting or restricting circumvention of such measures.
>
> When you convey a covered work, you waive any legal power to forbid circumvention of technological measures to the extent such circumvention is effected by exercising rights under this License with respect to the covered work, and you disclaim any intention to limit operation or modification of the work as a means of enforcing, against the work's users, your or third parties' legal rights to forbid circumvention of technological measures.

In other words, a redistributor under GPL 3 cannot simultaneously claim the enjoyment of the license and bring claims under this section of the DMCA; the two positions would be antithetical and doing so would, arguably, have been a prohibited "further restriction" under Section 7.

Disabling and Obfuscation[11]

Section 6 of GPL 3 contains provisions intended to solve the so-called Tivoization problem. Free software advocates were concerned that, while GPL required the delivery of source materials, it did not exactly require the redistributor to enable the recipient of the software to change and install the software in a product in which the software is embedded. On its face, it can be challenging to comply with GPL in a meaningful way for software in embedded systems. Suppose you distribute a toaster that includes GPL software. The software is not written or debugged on the toaster. It is written in a development environment, possibly including an emulator debugging tool that mimics the operation of the toaster but runs on a standard computer system. How can you give the recipient the means to revise the software? At a certain level, it may be impossible, or it might require you to deliver

[11] This is sometimes called the "Tivo problem." I make no claims about Tivo or its practices; this is simply a shorthand phrase common in the free software world.

prototype devices—which is clearly not a requirement of GPL. Most embedded system developers comply with GPL by providing source code that compiles in a standard environment.

In addition, enabling users of physical devices to modify embedded software can cause serious support or security issues. Some vendors lock down their devices by introducing technical blocks that will disallow operation or connection to communications networks if the software has been changed. This is not a problem with an obvious solution. How can the freedom of the recipient to change software be balanced against the need to prevent software bugs that would cause physical damage or security breaches?

Section 6 only applies to certain kinds of products. It defines a *User Product* as (1) a consumer product or (2) anything designed or sold for incorporation into a dwelling.[12] It requires the distributor to provide installation information.

> "Installation Information" for a User Product means any methods, procedures, authorization keys, or other information required to install and execute modified versions of a covered work in that User Product from a modified version of its Corresponding Source. The information must suffice to ensure that the continued functioning of the modified object code is in no case prevented or interfered with solely because modification has been made.
>
> The requirement to provide Installation Information does not include a requirement to continue to provide support service, warranty, or updates for a work that has been modified or installed by the recipient, or for the User Product in which it has been modified or installed. Access to a network may be denied when the modification itself materially and adversely affects the operation of the network or violates the rules and protocols for communication across the network.

Notwithstanding the carve-out, which was added to allay concerns expressed by many parties during the drafting discussions, many consumer electronics companies are reluctant to use code under GPL 3.

[12] Note that the focus is on the product rather than the user; the open source definition disallows making distinctions between users. See www.opensource.org/docs/definition.php.

They find delivering this information to be too potentially risky to their business.

Affero GPL

Because the threshold for copyleft triggers did not change in the move from GPL 2 to GPL 3, FSF released a variation of GPL 3 to close this loophole. Before that time, several licenses had set SaaS use as a threshold for triggering copyleft issues. The most notable of these was the Affero GPL, which was created by Affero Inc. The version of Affero GPL that existed prior to GPL 3 said this:

> If the Program as you received it is intended to interact with users through a computer network and if, in the version you received, any user interacting with the Program was given the opportunity to request transmission to that user of the Program's complete source code, you must not remove that facility from your modified version of the Program or work based on the Program, and must offer an equivalent opportunity for all users interacting with your Program through a computer network to request immediate transmission by HTTP of the complete source code of your modified version or other derivative work.[13]

FSF incorporated this idea into a variant of GPL called Affero GPL 3, which is identical to GPL 3 except that making the software available over a network triggers the copyleft conditions that require source code to be made available. It contains this provision:

> Remote Network Interaction; Use with the GNU General Public License.
>
> Notwithstanding any other provision of this License, if you modify the Program, your modified version must prominently offer all users interacting with it remotely through a computer network (if your version supports such interaction) an opportunity to receive the Corresponding Source of your version by providing access to the Corresponding Source from a network server at no charge, through some standard or customary means of facilitating copying of software. This Corresponding Source shall include the Corresponding Source for any work covered by version

[13] www.affero.org/oagpl.html.

> 3 of the GNU General Public License that is incorporated pursuant to the following paragraph.

Note that these conditions only apply if you modify the software. Mere users need not make source code available. In part because this license is not very commonly used, it is still unclear what constitutes *interaction*. For instance, if this license covered a language engine and you made changes to that engine and made a program executing in that language available to users online, would the language engine be interacting remotely with the users? Or does this require direct interaction, such as with a GUI or application? No one knows.

For many years, the most popular program available under AGPL was MongoDB, a database engine. MongoDB is available under dual licensing, so anyone who is worried about this issue simply buys an alternate license. However, in 2018, MongoDB moved to the newly-drafted SSPL, discussed in detail in Chapter 22: Recent Developments.

Affero GPL 3 also says this:

> Notwithstanding any other provision of this License, you have permission to link or combine any covered work with a work licensed under version 3 of the GNU General Public License into a single combined work, and to convey the resulting work. The terms of this License will continue to apply to the part which is the covered work, but the work with which it is combined will remain governed by version 3 of the GNU General Public License.

GPL 3 contains a counterpart in Section 13:

> Use with the GNU Affero General Public License.
>
> Notwithstanding any other provision of this License, you have permission to link or combine any covered work with a work licensed under version 3 of the GNU Affero General Public License into a single combined work, and to convey the resulting work. The terms of this License will continue to apply to the part which is the covered work, but the special requirements of the GNU Affero General Public License, section 13, concerning interaction through a network will apply to the combination as such.

This allows the two licenses to be horizontally compatible; it's possible to combine GPL 3 code and Affero GPL 3 code in a single program.

Apache 2 License Compatibility[14]

After the kerfuffle about whether Apache 2 and GPL 2 were compatible, the GPL 3 drafting project aimed to bury that hatchet. Apache 2 and GPL 3 were declared compatible—though given that many people did not understand the compatibility argument in the first place, the solution to the problem seemed a bit too easy. In any case, FSF apparently determined that the addition of patent and other provisions to GPL 3 reconciled the two.[15]

[14] See Chapter 4: License Compatibility.

[15] www.gnu.org/philosophy/license-list.html#apache2.

Chapter 11

Open Source Policies

Today, it is considered a best practice for a company to have a written open source policy. In some areas of law (such as employment law), the fact of having a written policy has a direct bearing on liability, but for open source, the policy is entirely functional. Copyright infringement is largely a *strict liability* regime—meaning that intent is not an element of infringement.[1] Therefore, a company will not be able to avoid liability for the infringing acts of its employees merely because it has a written policy. A written policy that everyone disregards is worthless and, in fact, can be counterproductive because it sends a signal that the company will look the other way when violations occur.

Written policies need not be long or complicated. But having them in writing is useful, particularly when a company has multiple development teams who may be in different countries and may speak different languages. Having a written policy gives these teams the time and opportunity to read and understand the policy, ask questions, and refer to the policy when necessary.

However, there is no one-size-fits-all policy. Here are some things to consider if you are contemplating an open source policy for your business.

Start Small

A robust open source policy usually covers a few topics: compliance processes, code release and contribution, and baseline

[1] Willful infringement can lead to a higher statutory damages award, but that is a relatively minor enhancement compared to the equivalent in patent law.

requirements for vendor contracts and other transactions. A good model policy that covers basic compliance topics is available from the Blue Oak Council at https://blueoakcouncil.org/company-policy. A more robust model policy is available on my web site. Please see the last section of this book for instructions on how to get useful policy examples.

Your company may not need all the topics at the outset. Most companies start with compliance. Topics like contributions, code releases, and transactional terms can be added when the company is ready for them; before that time, these elements are usually handled ad hoc.

Business Processes

The corollary to the policy is the process that attends it. The policy only describes certain checks and processes; the business then has to implement them. For example, if using GPL code in a product requires the approval of an engineering manager or the legal department, the company may want to implement an automated way to generate that request. Also, every company needs to consider how it will capture and store the information about the open source software it is using. Using spreadsheets is a fairly bad way to do this, and any method is only useful if it is easy to update and if updating it is incorporated into the development and release cycle. A company should also consider when in the development cycle the open source "seal of approval" needs to be applied. The closer to release it happens, the more likely that any remediation or substitution of code to comply with open source licenses will delay release or simply be abandoned in favor of meeting the release date.

Most contemporary open source compliance tools, like FOSSA or Black Duck, can help you to automate this process. The principal improvement in open source compliance tooling in recent years is better user interaction and focus on implementing the process, not just identifying problems.

Staffing

The title of the person designated to approve requests to use open source software in the business will differ from company to company. Some companies want to conserve the resources of their legal department and delegate some decisions to engineering management; some take the opposite approach and delegate the question to legal; some have a committee to deal with open source matters (usually composed of representatives from legal, engineering, and management areas); some hire a specialist paralegal or lawyer to handle open source matters. Most open source compliance matters can be handled by someone with minimal, but solid, engineering and legal training. As the volume of approvals at a company goes up, decisions need to be pushed down to an operational level so senior management is not tasked with making routine or recurring decisions. Recurring requests to use the same open source software are frequent because code that works well in one company product may work well in all of them.

License-Based Review

The model policy focuses on review of software based on the license that covers it, and this is the way most companies review their potential use of open source software. However, there are good reasons to take a more nuanced approach. Some companies review and approve the use of open source software package by package; this is most common when the company has heightened security concerns or exacting product qualification requirements from their customers. Some companies create a "sandbox" from which all open source software used in the company must be drawn, and they forbid downloading directly from the Internet. Some companies review open source software for patent infringement or other legal matters (such as export restrictions) as well. Accordingly, the model policy takes a lightweight approach to review, based only on the license.

Use Cases

A policy for a SaaS company may not fit a consumer electronics company, and vice versa. They are at opposite ends of the open source risk spectrum. Depending on the company's appetite for risk, it may require much less review of open source risks than do other companies. However, SaaS companies should keep in mind that basing an open source compliance policy on the absence of distribution can lead to terrible results. Most SaaS companies eventually distribute their products—whether to customers who want a private instance of the software to manage regulatory or security issues, to corporate partners or affiliates, or to successors in interest. So don't start what you can't finish—interleaving GPL code with proprietary code in the same program, particularly at a granular level, will likely lead to problems someday. Also, companies that assume they are immune to open source risk will be tempted to ignore record keeping. Taken together, these practices can mean the software can never be distributed. Then, when the company decides that distribution is its next business initiative, no one will want to deliver the news that it can't be done without a massive re-engineering and audit effort.

Chapter 12

Audits and Due Diligence

While most audits are conducted at the time of an investment, corporate sale, or other such transaction, other audits are conducted in connection with customer sales or simply as a process of establishing responsible internal business controls that manage intellectual property hygiene. Today, many customers who are buying licenses for software insist on a full disclosure of open source components. In fact, if the product is to be distributed to the customer, the vendor has an obligation to provide this information to the customer, in the form of the license notices required by the applicable open source licenses. So, a company should be prepared at all times to disclose the third-party open source software in its products. It is best to be ready for this challenge before you need to make guarantees that your company is compliant.

The Challenges of Open Source Compliance

Open source compliance is not especially difficult—particularly compared to compliance requirements for proprietary licenses, which might require constant tracking of the software's users, servers, or use cases. In fact, on a conceptual level, open source compliance is easy in most cases. Only a small number of use cases raise the potentially complex legal issues discussed in Chapter 6: What Is Distribution? and Chapter 8: The GPL 2 Border Dispute. But open source compliance does involve some information management challenges, and when people say open source compliance is difficult, they are usually referring to this need to track and disclose information.

The information management challenge arises because open source software tends to circumvent business processes. In contrast, if you must license software from a proprietary vendor, then you must first pay a license fee. Most businesses have internal controls that capture the payment of fees: if you must write a check to a licensor, you must sign a license agreement, and the license agreement must undergo legal review, which causes a lawyer or contract negotiator to consider whether the uses contemplated by the company will be compliant and whether the fees to be paid are reasonable in light of the licensing terms. However, open source software is always free of charge and can usually be downloaded freely from the Internet without signing any agreement or paying any fee. This means that for most organizations, there are no internal controls that capture the initial use of the software and, therefore, there is no advance consideration of whether the company can comply with the licenses. Open source compliance is left as a "cleanup" matter for a later time.

As a consequence, many companies wake up one day to discover that they do not know what open source software they are using or whether they are compliant with open source licenses. When this happens, companies may initiate an audit. Alternatively, companies are often forced to initiate an audit when a potential buyer or customer demands it.

Snapshots, Surveys, and Title Searches

There are two kinds of audits—the snapshot and the protocol. If you have been asked to undertake an audit to support a transaction like an investment or acquisition, you are probably taking a snapshot. The snapshot identifies the open source software in your code base at a particular moment in time; for an acquisition, that would be the moment of closing the transaction. However, companies that are more sophisticated about open source compliance want to make sure that they are always compliant—not just when someone else demands it. Therefore, they conduct diligence on an ongoing basis as code is added or changed in the code base of their products, and for this they need to

implement business protocols to collect and manage open source licensing information.

An open source audit is like any other audit in the sense that one is looking for problems, but one can never be sure that all problems have been found. The art of auditing involves engaging in efforts that are financially and technically practical to eliminate the most significant problems. The proper level of effort depends on how risky you think a mistake will be, and that in turn depends on the use case for the software you are auditing and your assessment of the existing internal controls used to manage open source licensing compliance.

There is no one right way to conduct an audit. Lawyers are often asked to make a decision about whether to hire a code-scanning consultant to examine the code. It is never wrong to do this. However, hiring a consultant is like buying insurance: it may be useful, but it may not always be a good business decision. It's possible to spend all the money you like on managing risk, but it only makes sense to spend money on activities that actually reduce measurable risk.

Economically speaking, the equation is this:

```
AC < R * L,
```

where AC = Audit Cost, R = Risk of a problem, and L = Loss that would occur if the problem occurred.

Thus, a problem with a small likely loss is not worth spending a great deal of money to find. Risk and loss depend on use case and your assessment of a company's internal controls. If a company has no internal controls on open source software use, R will be higher. If the software is distributed in a consumer electronics product, both R and L will be higher than they would be for a SaaS product.

It is not difficult to tell whether a company has internal controls for open source compliance processes in place—all you have to do is ask. For instance, if you ask for a list of open source code that is in a product, and if the company doesn't have such a list (or doesn't understand the question), then the company likely has no compliance process in place. In such a case, hiring a code scanner is essential. However, if the company provides a list of open source code and licenses that looks

reasonably professional, you may decide that hiring a code scanner is unlikely to yield additional information that is material to risk.

There are two overarching approaches to automated auditing: forensic matching and string searching. Companies like Black Duck Software (now owned by Synopsys), FOSSA, and Palamida (now owned by Flexera) perform services known as forensic matching. In other words, they examine the source code for a particular code base and match it against known open source code. They identify the known open source code by examining a proprietary database that serves as "ground truth" for the review. The other approach is to run a string-matching program, which programmers often call a *GREP*.[1] This kind of program compares *strings* (which in programming language means text characters) in the source code against *regular expressions*—meaning target patterns, sometimes with wild cards—to find indicia of licensing terms such as copyright notices and licensing statements. Using GREPs is like typing a query into a search engine. For instance, if you searched the code base for "copyright," "Copr.," or "license," you would find copyright notices, licensing notices, and probably a lot of extraneous items as well. Programs like FOSSology, an open source utility for compliance, use this approach. They do not compare against any ground truth data.

Alternatively, you may be asked to conduct an audit based on the records kept by the company. Most often, such self-disclosures are in the form of spreadsheets or text lists, and they list software and licenses. The difference between these two approaches is like the difference between a land survey and a title search. Forensic matching is like a survey—it looks at what is actually there and tries to map it onto known information. String matching or self-reporting is like a title search—it looks at what people have recorded about the licensing of the code and assumes that the information is reliable.

In truth, neither of these approaches is foolproof, nor is either without its challenges. Forensic matching is less prone to false

[1] GREP, the name of an old UNIX routine that does string searching, stands for "global regular expression print."

negatives—it can find open source for which the copyright and licensing notices have been removed from the software being audited. This removal of licensing notices is considered poor programming practice, but it happens. However, forensic matching is prone to false positives in that it finds potential problems that are not actual problems. On the other hand, a GREP approach will not identify code for which licensing information has been removed or not properly preserved. Therefore, the GREP approach is very useful at identifying entire packages, files, or libraries of open source code, but it may not be successful in identifying fragments or snippets of code.

In an auditing effort, false negatives are more dangerous than false positives. Therefore, proper auditing techniques should tend toward zero false negatives and minimal false positives. To understand better how this works, consider the following seeming paradox: Most people who test positive for a serious disease probably don't have it. Does this mean the test is wrong? No, it means the test is properly designed to avoid false negatives. If the test is positive, more accurate tests will be done, and these tests are intended to weed out the false positives. The first level of testing is performed on a much larger population than subsequent tests, so it produces many more false positives than the true positives found in later testing. Any kind of audit of testing works in this way.

But as a result, reports that are generated by auditing efforts can be maddeningly difficult to review. They tend to include a lot of false positives, or "noise." Every code base will have small fragments of code that may be taken from elsewhere, but not all such fragments constitute infringement. For very common code lines, it may be difficult to tell the original source. Also, the copyright law will not protect what it calls *de minimis* use of copyrighted material.

The upshot of all this is that audit reports give you information, not answers.

Setting the Rules

Once you perform such an audit, you must determine whether the information you have found meets the criteria that you have set for compliance. For many companies, these criteria are embodied in an open source software policy. (For more on open source policies, see Chapter 11: Open Source Policies.) Different companies have different levels of tolerance for risk, and different use cases have different risk profiles. However, one thing is certain: you cannot tell whether an audit has passed your criteria if you don't have criteria to apply.

Today, most companies conducting audits consider permissive licenses to be of little concern. Of course, distributing code that includes open source software licensed under permissive licenses requires the delivery of a notice, so some basic compliance is always necessary. However, delivering notices is almost always possible, even if administratively burdensome. In audits, one is mostly looking for copyleft software—particularly copyleft software under licenses like GPL and LGPL—because of concerns that the audited software has been engineered in a way that causes compliance to be impossible, even if all of the information is known and the license notices are applied. For example, if a proprietary application uses a GPL library, it may be impossible to comply simultaneously with both licenses.

The Perils of Self-Disclosures

Self-disclosures are often skeletal, inaccurate, and misleading. They generally contain correct information about the open source code that is included—at least at a package level. A list of open source components can usually be generated by the build instructions for the software. This package-level information is reasonably reliable because the compiler must be given this information in order to build the product. But this information does not include any license information. In a typical open source self-disclosure, license information is prepared manually. However, it is often very inaccurate due to any or all of the following issues:

- Lack of license information
- No versions for licenses
- Conflicting license information
- Lists software that is not open source

Here is a hypothetical self-disclosure:

Linux...GPL

Jboss...LGPL

GCC...GPL

XYZ...GPL/MIT

Java ...GPL

Adobe...public domain

The problems with this self-disclosure are that there are no license version numbers, one of the items is clearly inaccurate (Adobe is not a specific piece of software and is likely neither open source nor public domain), one item references a dual license without explanation (which usually, but not always, means the licensor has a choice), one item (GCC) is a development tool that is unlikely to be in a product, and Java is generally under GPL+ FOSS exception. Correcting self-disclosures is a long-standing pastime in auditing. Those who do audits for a living see all manner of weird self-disclosure. One of my personal favorites was "GNU BSD." Therefore, if you are conducting an audit based on a self-disclosure, you should not rely on the licensing information you have been given unless you trust that the source of the information knows how to locate and vet the information.

This means that those conducting audits spend a lot of time researching licensing terms, starting from a list of packages. Researching licensing terms can be tricky. For instance, if you have been told that a code base includes a piece of software called FOOBAR, your first step would be to search for FOOBAR in a search engine. If you're lucky, you will find a FOOBAR project page that indicates the license terms. At this stage, several things can go wrong: there may be more than one project called FOOBAR, there may be no project called

FOOBAR, or the licensing information for FOOBAR may not be clearly indicated on the project web page. Moreover, if you take the trouble to download the FOOBAR source code, you may find that the licensing notices in the project are not the ones that are indicated on the project web page.

Therefore, researching licensing terms is more of an art than a science. Projects that are competently run will clearly indicate licensing terms on the web page, and the licensing terms will match the license notices that are actually in the code when you download it. Projects that are not competently run may have no license terms or inconsistent license terms. The cloud development platform GITHUB is especially known for projects with no licensing terms. Unfortunately, if no licensing terms are applied, the default is that no rights are granted to use the software. In such cases, there is no way to be compliant.

If you are presented with conflicting licensing terms, such as one license identified on the web page for a project and another in the source code COPYING file, you should make a reasoned judgment as to which set of terms is right. If a project seems not to have taken proper care with upstream rights, you may need to try to determine where the code came from to determine which licenses must apply. Generally, the license that is actually in the source code will be the more reliable indicator. But there is no magic to this inquiry. Assuming the licensor has made his own due diligence mistakes, you can rely on any license that the author has applied to the software. Placing a license notice on the software is usually enough to indicate that the author has intended to apply those licensing terms.

Versioning

Occasionally, the license terms that are provided to you in an audit process may be inaccurate because the project has changed its license terms. In other words, the license in the software notices for the code you are auditing may be correct, but it may not match the license terms in the currently available version of the software. One way to avoid this problem is to ask for version numbers in the disclosure to be audited.

License changes almost always coincide with the release of a major update, so it is usually possible to check the licensing from publicly available sources if you have the software version number. It is most common for a project to change to more permissive licensing terms rather than less permissive licensing terms, so this phenomenon does not usually cause audit problems. In fact, if the license that applies to the version of the software in the audited code base is too restrictive, it is possible that moving to a later release will solve the problem.

Solving Problems

If you identify a problem in an audit, you should always think about a solution. (This comment is mostly for the benefit of lawyers. Engineers always think about solutions; lawyers sometimes seem to think their job is to find problems and stop there.) Almost every audit problem has either an engineering solution or a licensing solution, or both. Always consider all of these possibilities:

- **Remove.** In any code base, quite a bit of code goes unused. There are various explanations, but the most common is that the code was used in testing but not in the final product. The actual binary of a software product need not contain all the code in the source code repository from which the product is built. Therefore, if you find a licensing problem, your first questions should be "Is this code really in the product? If so, can we remove it?"
- **Replace.** Open source packages are often created as alternatives to popular proprietary software, or vice versa. Therefore, if the open source terms are a problem, you may be able to find a proprietary alternative that you can substitute for the open source package with minimal engineering effort. Often, proprietary substitutes are not expensive—after all, there is a substitute good that is free of charge.
- **Re-engineer.** This is the option everyone wants to avoid because it is expensive and time-consuming. Of course, it is rarely

necessary unless the issue involves GPL or LGPL, because these are the licenses whose conditions rely on the means of integration of software.

- **Relicense.** If the open source software is provided under license terms that you cannot comply with for the use case you are auditing, you may ask the author if he or she is willing to grant an alternative license. Clearly, this approach will not work with most major projects, such as the Linux kernel or Apache web server, which have set licensing terms based on the project's philosophy. Relicensing works best for individual authors. Keep in mind that if a project has more than one author and you seek an alternative license, you may have to locate and make an agreement with all the authors.

Audits and Mergers & Acquisitions

Anyone who has ever been involved in a mergers and acquisitions (M&A) transaction knows that open source audits can be frustrating to accomplish within the confines of such a transaction. Despite the fact that it is more popular than ever to conduct code scans for M&A deals, most due diligence in M&A is still based on self-disclosure. Unfortunately, these disclosures tend to come late in the deal process and be extremely inaccurate, so they tend to create issues too late to do much about them. If you are running a startup business, you can effectively streamline investment transactions and corporate sales by engaging in at least basic open source compliance and keeping records of the open source software you are using. Pre-auditing your own products can save you a lot of time and trouble and make your company look more professional in the due diligence process. (For more on this, see Chapter 17: Mergers & Acquisitions and Other Transactions.)

Going the Extra Mile: Process

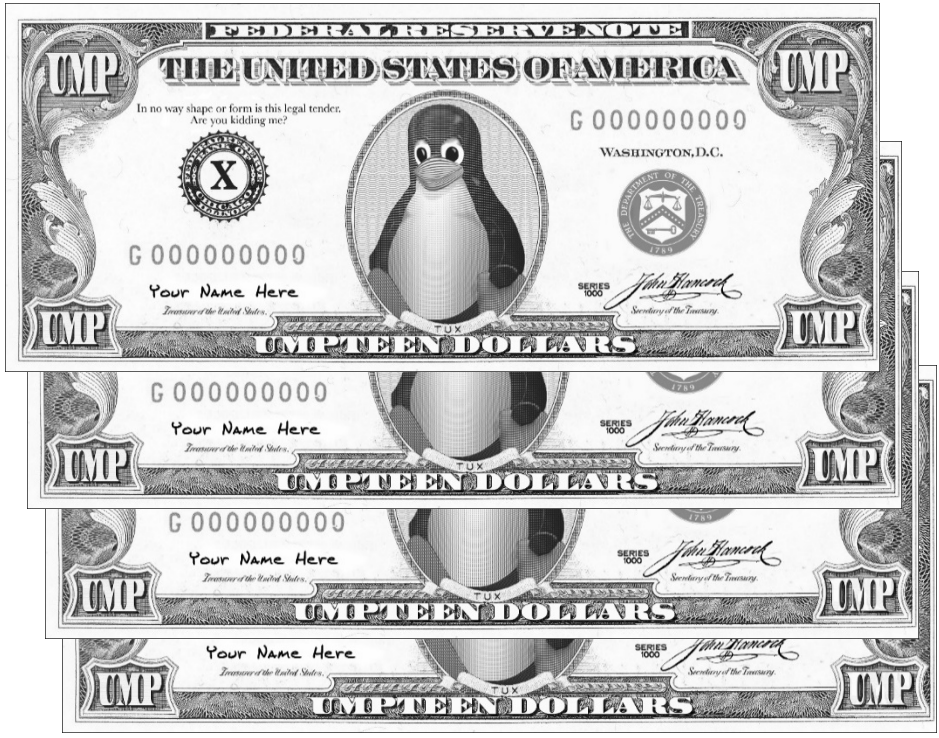
Some of this detail can be finessed if the target complies with a standard called OpenChain, which identifies the key requirements of a proper open source compliance program. The specification is short,

simple, and easy to read, and covers topics like having a written policy and developing expertise within your organization, a process for identifying a bill of materials, and basic compliance processes.[2] OpenChain allows adopters flexibility to choose processes to meet its requirements. OpenChain is a process description, and M&A representations mostly concern the results of the process.

[2] https://wiki.linuxfoundation.org/_media/openchain/openchainspec-2.0.pdf.

Part IV

Intersection with Patents and Trademarks



Chapter 13

Open Source and Patents (Grants, Defensive Termination)

Software patents come with plenty of political baggage—the open source community is vehemently anti-patent. But whether software patents are good policy or bad policy, they are part of our law.[1] And because it can be hard to find objective and practical analysis of the intersection between open source software and patents, businesspeople need a framework in which to make rational decisions about software patents based on the current rule of law.

This chapter covers two topics. The first is the question of how third-party patents—patents owned by parties who are not engaged in open source licensing—can affect the community's freedom to use open source software. The second is the question of how open source licenses affect the patents of those who are engaged in open source

[1] Many people are fond of saying software patents only exist in the United States, but that is not quite accurate. In the United States, it is not so much that software patents exist as that they have not been excluded, because Congress did not so limit the patent law ("Whoever invents or discovers any new and useful process, machine, manufacture, or composition of matter, or any new and useful improvement thereof, may obtain a patent therefor, subject to the conditions and requirements of this title" (35 USC. § 101)), which can cover "everything under the sun made by man" (*Diamond v. Chakrabarty*, 447 US 303 (1980)). Only Congress has the power to change this law. So although software patents are less common in Europe than in the United States, the difficulty of drawing a bright line excluding software patents plagues both sets of law. The European Patent Convention (EPC), Article 52, paragraph 2, excludes "programs for computers" from patentability, but paragraph 3 says, "The provisions of paragraph 2 shall exclude patentability of the subject-matter or activities referred to in that provision only to the extent to which a European patent application or European patent relates to such subject-matter or activities as such." However, the EPO considers an invention patentable if it provides a new and non-obvious "technical" solution to a technical problem, and that invention can be embodied in software.

licensing. The first is largely a policy question. The second is a question of license interpretation and intellectual property strategy.

The Great Patent Debate[2]

The Internet teems with discussion about whether software patents pose a particular threat to open source—as opposed to a threat to software in general—as well as about whether software patents are right or wrong. Some of this discussion is interesting and thoughtful, and some of it is laced with profanity and rage (and therefore may be interesting but not particularly useful). But one thing is clear: the open source community hates software patents. For instance, the preamble to the GPL contains a statement that "any free program is threatened constantly by software patents," and the FSF and other free software organizations actively engage in lobbying and other efforts to defeat particular patents or software patents in general.

Whether or not you agree that software patents are immoral, you should understand that this attitude is nearly ubiquitous among today's software engineers. In practical terms, therefore, it can be difficult to involve software engineers in patent portfolio management. Historically, patent lawyers worked closely with engineers to prepare patent disclosures and applications; today, some engineers refuse to engage in the process. In 2012, Twitter published an "Innovator's Patent Agreement"—a pledge to inventors that their employer will not use their patents for offensive purposes:[3]

> It is a commitment from Twitter to our employees that patents can only be used for defensive purposes. We will not use the patents from employees' inventions in offensive litigation without their permission. What's more, this control flows with the patents, so if we sold them to others, they could only use them as the inventor intended.

[2] An earlier version of this discussion appeared in "The Fuzzy Software Patent Debate Rages On," *Linux Insider*, February 25, 2005.

[3] http://github.com/twitter/innovators-patent-agreement.

Other companies have also signed on to an informal pledge saying: "No first use of patents against companies with less than 25 people."[4]

Reading between the lines, companies may think that unless they promise not to use their patents for offensive purposes, they won't be able to retain the best and brightest engineers or secure their assistance with the company's patent prosecution efforts. Many other companies have made similar decisions or promises—usually less publicly. In technology companies, engineers are the star talent, and management caters to the political leanings of the stars.

More or Less at Risk?

The open source community will certainly not resolve the larger patent debate, because the policy decision to limit the scope of patentable subject matter goes far beyond open source software. But there is a narrower question—which is perhaps more interesting because it is more likely to be answered—as to whether open source software is more or less vulnerable to claims of patent infringement than is proprietary software.

GPL 2 says, "Any free program is threatened constantly by software patents." But there is a difference between being threatened and being thwarted. The question of whether open source is particularly risky to use—in the sense that it is vulnerable to patent claims—is of great practical importance to those using open source software in business. This is an open source issue, not just a free software issue. But the ability of a third party to bring patent claims has nothing to do with the outbound license terms applied to software—and that, in part, is the key to answering this question.

One of the most difficult concepts in intellectual property law is this: patents are rights, and nothing but rights. Owning a patent gives the owner nothing but the right to exclude others from practicing the invention claimed in the patent. A patent is sometimes referred to as a *non-enabling right*. It does not enable its owner to make any product or

[4] www.thepatentpledge.org.

engage in any business. Enforcing a patent by bringing a patent infringement suit does not require the patent owner to engage in any business he is trying to protect. This is why patent "trolls" can exist—companies that do not engage in any business except suing others for patent infringement. Patents are negative rights only.

Under copyright law, in contrast, an author has to actually create something to own the rights to it. The same is true under the other intellectual property regimes—trade secret and trademark; in each case, one must actually create something to own the rights to it. Trade secrets arise because someone has created confidential business information. Trademarks arise because someone has used them in trade to designate the source or origin of goods.

This non-enabling nature of patent rights is part of what makes people hate patents. They view patents—not unreasonably—as being a net loss for society because their owners can stop others from engaging in innovation without engaging in any themselves. But the premise of patent law is that the publication of the patent is intended to advance the useful arts by disclosing inventions—making them "patent"—to the world. Perhaps the reason the patent system seems broken today is that people no longer consider patents to be a good means of teaching innovation. This, in turn, probably stems from a perceived failure of the patent office to limit patents to inventions that are truly innovative and not obviously based on the state of the art. When half of the patents that are enforced are invalidated, mostly due to obviousness, it's hard to disagree.

Moreover, under other intellectual property regimes, independent authorship is a defense to a claim of infringement. Under copyright and trade secret, if one takes the time and trouble to reinvent the wheel without copying or using the proprietary material of others, one can use the wheel. The fact that someone else used a wheel first makes no difference, though the defendant may be hard-pressed to prove that he created the idea of the wheel independently rather than mimicking it. Independent invention is not a defense to patent infringement, and as a result, one can easily infringe a patent without knowing it, even if one has created the invention without "stealing" others' ideas.

The theory is that an issued patent, which is free for anyone to read and learn from, is constructive notice to the world of the invention and how to practice it. The word *patent* means open for inspection, but in practice, searching for a patent that covers an invention one might want to practice is expensive, time-consuming, and complex—and therefore functionally impossible. In fact, the law creates a disincentive for doing this research (sometimes known as a freedom-to-operate study) because being aware of potential patent infringement only exposes one to "willful" infringement, which makes one potentially liable for triple damages and attorneys' fees. So in sum, patent law has become a minefield, and the winners often seem to be companies that don't innovate at all.

As a consequence of the non-enabling nature of patent rights, software can be developed in any way and still be vulnerable to patent claims. Even code written completely from scratch, line by line—if any such code exists—could infringe a patent, and the developer would not know this until he got sued. That code could be proprietary or open source—it doesn't matter.

So the GPL's assertion that all free software is constantly threatened by software patents is true—but it is only part of the truth. Any software is constantly threatened by software patents, and yet we still have software. In fact, open source software and software patents have grown up together; the decision in *Diamond v. Diehr*, 450 US 175 (a landmark case allowing a software patent), was handed down in 1981; GPL was written in 1991; and the decision in *State Street Bank v. Signature Financial Group,* 149 F.3d 1368 (allowing a business method patent) was handed down in 1998. Both software patents (and their sisters, business method patents) and open source have enjoyed tremendous success in the 1990s and the early twenty-first century.[5]

If we peel back another layer of this onion, the question of whether open source software is particularly vulnerable to software patents becomes more interesting. Those of you who have negotiated software

[5] For a very good summary history of software patentability in the United States, see www.bitlaw.com/software-patent/history.html.

agreements know that, given the impossibility of knowing whether software infringes third-party software patents, liability for patent infringement is a difficult issue to address in software transactions. Neither side—licensor/licensee nor assignor/assignee—wants to bear this uncontrollable and potentially expensive burden. A pure open source software license is different from a proprietary license in this respect. Software provided under a pure open source license is provided as-is, with no warranty, and that means the licensor is not bearing any risk for claims of patent infringement. Proprietary software is often—though not always—supported by a licensor indemnity covering patent infringement claims.

But this is not as simple as it seems. A great deal of proprietary software comes with no indemnity for patent infringement. (Next time you click to accept an end user license, you may see an example of this.) Or perhaps there is an indemnity limited to the price of the software, which will be inadequate compared to defending a patent infringement claim. Even preparing a simple response to a patent infringement lawsuit costs many thousands in legal fees. So it's not true that proprietary software always comes with an indemnity.

And sometimes, open source software does. Above, I referred to pure open source licensing, meaning getting software under an open source license with no additional business terms. Although the license itself provides no indemnity, a vendor whose products include open source software can agree to a contract for an indemnity, usually as a quid pro quo for maintenance and support or services fees. The important thing to remember about indemnities is that they have an economic cost, so no one gives them for free. Open source software is more likely to be free, so it frequently comes without an indemnity. (For more detail on this question and what is reasonable and customary for the bearing of risk in commercial transactions, see Chapter 17: Mergers & Acquisitions and Other Transactions.)

To understand the risk of patent infringement liability, we need to consider the motivations behind patent infringement claims. If patent infringement exists, nothing will come of it unless a claim is brought by the owner of the patent. To assess the risk that a patent owner will bring

a claim, we must understand why the owner would want to do so. After all, patents are expensive to prosecute; filing an initial application for a software patent costs $20,000 on average in legal fees, plus additional legal fees to respond to the objections of the patent examiners before the claims are actually allowed and the patent issues. So no one gets patents just for fun. In addition, patent infringement claims are not only expensive to defend—they are expensive to bring. So why spend all that money?

The three reasons to own patents and thus pay the attendant fees are:

- suing people for money,
- counteroffensive claims, and
- disrupting competitors.

Money

The first reason is the most straightforward: a plaintiff in a patent action can sue for monetary damages. Patent infringement claims are economic weapons. A lawyer joke: rule number one is don't sue anyone poor. Open source projects make bad targets for damages claims because they usually do not have any money.[6] Companies using open source software to make money, however, can be more promising targets.

As for the amount of money one can hope to get by suing, patent damages can be complex to calculate, and litigation over the methods used to calculate them is often fierce. Damages are (a) at a minimum, a reasonable royalty rate for a license to the patents (35 USC 284),[7] and (b) lost profits of the patent holder.[8] Neither of these measures is particularly attractive in the open source landscape because open source

[6] In light of this, claims against open source projects are rare. A notable exception is *Rothschild Patent Imaging LLC v. GNOME Foundation*, N.D. Ca 3:19-cv-05414 Filed 08/28/19.

[7] *Georgia-Pacific Corp. v. United States Plywood Corp.*, 318 F. Supp. 1116 (S.D.N.Y. 1970), mod. and afford, 446 F.2d 295 (2d Cir. 1971), cert. denied, 404 US 870 (1971).

[8] *Panduit Corp. v. Stahlin Bros. Fibre Works, Inc.*, 575 F.2d 1152, 197 U.S.P. Q. 726 (6th Cir. 1978).

software tends toward backbone, commoditized, or unprofitable functionality, where there is not much room for profit and therefore not much room for patent royalties.

Counteroffensives

Counteroffensives are key motivators for patent enforcement. Most companies that own patents do not engage in proactive enforcement (also known as *licensing programs*); they build *defensive patent portfolios* to enforce in the event a patent suit is brought against them. Before the rise of the patent troll, this was the norm; many large companies entered into cross-licenses that provided a kind of nuclear stalemate discouraging patent claims among big players. Anyone who did not participate hoped that the value of its own patent portfolio was enough to discourage third-party claims.

Counteroffensive claims are common in proprietary software, but not in open source. That is because most companies with proprietary products seek patent protection, but open source projects rarely have patent rights to enforce. If the open source provider does not bring a patent claim, there is no reason for a counteroffensive.

A counterexample is a community defense program such as the Open Invention Network (OIN).[9] Such programs seek to build patent portfolios to defend open source. This requires obligating patent owners such as proprietary software companies, who may have patents with useful counteroffensive claims, to bring lawsuits in response to a claim accusing an open source project, even if the claim is not brought against the patent owner itself. But gauging the success of such efforts is difficult, because for them success is the absence of lawsuits. Many people are skeptical that these pledges to "defend" Linux are effective—though they do result in some useful publicity for their supporters.

However, when people complain about patent enforcement, they are not usually talking about defensive enforcement. Indeed, as

[9] The OIN involves both a patent pool and defensive pledges. See www.openinventionnetwork.com/about.php.

described in the second part of this chapter, even some open source licenses acknowledge the necessity of bringing defensive claims—even if those claims accuse open source software.

Competitive Disruption

The final motivation for bringing a patent infringement claim is to disrupt competition. On its face, this would be a likely motivation for a patent claim in the open source landscape; if a company is competing with an open source product, it has potentially serious competitive concerns, and its motivation to levy patent rents on competing open source product is high. After all, open source software is free of license fees and can therefore easily undercut sales of competing proprietary products. For example, Microsoft engaged in an aggressive and successful program of patent licensing for embedded Linux and Android products, presumably to disrupt competition with its Windows and Windows mobile platforms, before joining Open Invention Network in 2018.

However, in practice, very few companies bring patent claims accusing open source software. It is wildly unpopular to bring such claims and doing so is a scorched-earth public relations strategy. Leaving aside trolls (who don't need goodwill because they have no products to protect), only two companies have been active in patent enforcement against open source software: Microsoft and Oracle. Both are so large and entrenched in their markets that they can afford some derision by the open source community.

On the other hand, the poster child for destroying a business by bringing an open source lawsuit is *SCO v. IBM*. Arguably, SCO was a dying company before it went on the warpath against IBM and Linux, but if it had any business to preserve, it put the last nail in its own coffin with this landmark lawsuit. The SCO case was not a patent lawsuit, but it was the first major litigation relating to open source. (See the discussion in Chapter 19: Enforcement and Obstacles to Enforcement.) If nothing else, it showed how spectacularly unpopular it was to bring a suit accusing open source software. The media circus that resulted did

not help SCO's case. Most major technology companies, including those with significant patent portfolios, are more interested in preserving their goodwill than in any gain they could realize from bringing patent lawsuits accusing open source software—even if those lawsuits might disrupt the businesses of their competitors.

Based on these assumptions about the motivation for bringing patent lawsuits, we can now turn again to the question of whether open source software is more or less vulnerable to patent challenges. Overall, my analysis shows that open source software is less vulnerable than proprietary software. However, the question is a difficult one with no simple answer. The reasoning is below.

Discovery

One significant difference between open source and proprietary software is that open source software is freely available to review. This means a potential patent plaintiff does not have to engage in the legal discovery process—asking the court to mandate delivery of evidence—to investigate whether potentially infringing software actually infringes the plaintiff's patent.

This easier means of identifying potential infringement cuts both ways for our question. On the one hand, if a potential plaintiff truly cannot tell whether proprietary software is practicing certain inventions, said potential plaintiff will be less likely to identify the infringement but more likely to bring an actual lawsuit to compel discovery. On the other hand, if a potential plaintiff can easily review potentially infringing software, said potential plaintiff may be more likely to determine that a claim has no merit or to bring the claims sooner rather than later.

This is probably, overall, favorable to the defendant. The longer software has been on the market, the more potential damages in an infringement suit and the harder it may be to engineer around the issue. Moreover, if accused software has been available in source code form for a long time, the defendant may claim that the plaintiff should have brought a suit sooner because it had constructive notice that the invention was being practiced. This is a legal doctrine called *estoppel* or

laches, which discourages plaintiffs from sitting on their rights and allowing infringement to accumulate in order to seek more damages.

Also, those analyzing this issue should beware of making simplistic assumptions about the ability to determine whether binary code is practicing certain inventions. Some software patents have claims that describe inventions at a very high level. There is no need to examine source code to determine whether software practices such inventions—only to know what the software does. Moreover, those skilled in the art may be able to examine binary code to determine what kind of functions it performs. Languages like Java can easily be decompiled. Software written in high-level scripting languages such as Ruby, Perl, or JavaScript is always in source code form, regardless of whether it is licensed under open source terms or proprietary terms. Don't just assume that anything distributed in executable form is hidden from curious eyes—it may be, but it may not be.

Workarounds

Just as the open source community can easily cooperate on fixing bugs and software, it can easily cooperate on engineering around patent claims. This happened, for example, in the *Jacobsen v. Katzer* case, in which volunteers from the community assisted Jacobsen in locating prior art and suggested re-engineering to avoid the patent claims brought by Katzer. It also occurred in *Rothschild Patent Imaging v. Gnome Foundation*.[10] Organizations such as Open Invention Network, the Electronic Frontier Foundation, and the Software Freedom Law Center sometimes coordinate volunteers to thwart patent claims that accuse open source software. Overall, this means that open source software, even if accused of patent infringement, can be more easily changed to avoid ongoing infringement and thereby reduce potential damages and avoid injunction.

[10] For a fuller discussion of these cases, see Chapter 19: Enforcement and Obstacles to Enforcement.

Obviousness

Approximately 50 percent of all patents are invalidated when they are enforced, and after the decision in *Alice Corp. v. CLS Bank International* in 2014, which narrowed the scope of patentable subject matter, software patents have suffered even more from enforcement in recent years. Mainly, invalidation happens when the defendant identifies prior art that was not disclosed during the patent application process and that renders the patented invention obvious in light of that prior art. The America Invents Act of 2011 also allows a defendant to bring a parallel action to challenge the validity of the patent (an interpartes review, or IPR), often while staying the case brought against the defendant in federal court.

One complaint about software patents is that there are not sufficient repositories of prior art to assist patent examiners in eliminating obvious patent claims.[11] This was a more compelling argument when software patents were a more recent development. However, in the 30 years since the *Diamond v. Diehr* case, a great deal of patent prior art in the software world has been developed. In any case, because open source software is freely available for anyone to examine, it is also a huge store of publicly available potential prior art. Open source software therefore has a tendency to reduce the ability to successfully prosecute patents. This does not protect open source software from patent claims per se, but in some cases, the open source software could predate the patent that is being used to accuse it. This would raise an obviousness defense that might easily defeat a patent claim.

No Champion

A countervailing argument—that open source software is riskier—is based on the premise that while proprietary products tend to have defensive patent portfolios to back them and that the consequent

[11] Also, patent examiners are overworked, and the US Patent and Trademark Office is underfunded, but that is a separate policy argument about which much has been written.

potential of counteroffensive claims discourages patent lawsuits, open source projects do not have defensive portfolios. See the discussion above regarding counteroffensive measures as a motivation to bring patent claims.

Patent Grants and Provisions in Open Source Licensing

Now we turn away from policy and toward licensing technicalities. Many open source licenses have patent licensing built into them. However, these licenses look quite different from traditional patent cross-licenses, and interpreting them requires an understanding of how and why such clauses were developed. Many who are learning about open source licensing find the patent terms of open source licenses hard to understand. While traditional patent licenses usually seek to slice the patent rights granted into the narrowest slivers possible, open source patent licenses are purposely broad—and some would argue, purposely vague. That said, most patent grants in open source licenses are roughly similar in scope and not so difficult to understand.

Implied Patent Licenses

Even in the absence of an express patent grant, all open source licenses grant some patent rights, by implication. Case law on the doctrine of *implied license* is vague and sparse. The doctrine broadly rests on the premise that it would be unfair for a patent owner to grant a copyright license for software and then sue the licensee for patent infringement for engaging in the activities already licensed.

Implied license is actually a pastiche of several legal doctrines: legal estoppel, equitable estoppel, and patent exhaustion (or *first sale* doctrine).[12] Reading the case law on this question can be challenging,

[12] Nimmer & Dodd, *Modern Licensing Law*, §§4:2–4:3 (2007). David B. Kagan gamely tries to harmonize the doctrines in this article: "Honey, I Shrunk the Patent Rights: How Implied Licenses and the Exhaustion Doctrine Limit Patent and Licensing Strategies,"

but it helps to keep in mind that courts may be reasoning backward to try to avoid unfair results and may be less concerned about distinguishing the doctrine under which they are righting wrongs. Under *Wang Lab. v. Mitsubishi Elecs. Am.*,[13] the federal circuit held that under legal estoppel, a licensor cannot use patent infringement claims to take away a right that the licensor has already granted to the licensee. *Equitable estoppel* allows a licensee to rely on actions of the licensor that cause the licensee to reasonably believe the licensee has a license to the patent.[14] The foundation of this doctrine is old,[15] but its application to software is too new to draw a bright line around the scope of an implied license. For instance, the federal circuit has said, "Generally, when a seller sells a product without restriction, it in effect promises the purchaser that in exchange for the price paid, it will not interfere with the purchaser's full enjoyment of the product purchased. The buyer has an implied license under any patents of the seller that dominate the product or any uses of the product to which the parties might reasonably contemplate the product will be put."[16] It is true that open source licensing is "without restriction," but this case was not about software, and generally no price is paid for an open source license.

To avoid licenses implied by equitable estoppel, a patent holder can communicate that a separate patent license is necessary. Doing this in open source licensing can be challenging because open source licenses do not contain any reservation of rights[17] and copyleft licenses prevent

c.ymcdn.com/sites/www.lesusacanada.org/resource/collection/B2D9101A-F6BA-4E58-AC7E-A805EA0E9B6E/Exhaustion-Implied-License.pdf.

[13] 103 F.3d 1571 (Fed. Cir. 1997).

[14] For a more thorough discussion, see "Potential Defenses of Implied Patent License Under the GPL" by Adam Pugh and Laura A. Majerus, available at www.fenwick.com/FenwickDocuments/potential_defenses.pdf.

[15] "Any language used by the owner of the patent, or any conduct on his part exhibited to another from which that other may properly infer that the owner consents to his use of the patent …constitutes a license." See *De Forest Radio*, 273 US 236 (1927).

[16] *Hewlett-Packard Co. v. Repeat-O-Type Stencil Mfg. Corp., Inc.*, 123 F.3d 1445 (Fed. Cir. 1997).

[17] There are some exceptions. Mozilla Public License 2.0 contains a reservation of rights in Section 2.3. Also, Creative Commons Zero, a public domain dedication, expressly disclaims the granting of any patent right (Section 4.a); see http://creativecommons.org/publicdomain/zero/1.0/legalcode.

placing "additional restrictions" on the exercise of the license. However, it is not clear under many open source licenses whether requiring a separate patent license would be such a restriction.[18] A patent license is more likely to be implied where no express patent license exists—so even for open source licenses that contain an express patent grant and no reservation of additional licenses, it is possible that no additional license will be implied.

Some of the basic principles of patent licensing depend on how we define certain terms in the patent license, such as these:

- Definition of licensed patents (capture)
    - Patent owner
    - Time period of capture (which may include forward capture)
    - Listed or particular patents
    - Geographic limitations
- Definition of products licensed (the object of the grant)
- Field or territory definitions for the license grant (scope limitations)

A full discussion of patent licensing is beyond the scope of this book, but a couple of these elements are key to understanding patent grants in open source licenses.

While traditional patent licenses often capture only a specific patent or list of patents, an open source patent grant capture always consists of any "necessary claims" of the patent owner. The concept of a necessary claim is an old one in patent licensing, particularly in standards licensing and patent pools. That is unsurprising because the purpose of a patent license in an open source license is to create a kind of "patent commons" covering the software.

A *patent claim* is the part of the patent that describes the invention covered by the patent. A *necessary claim* is a patent claim without which it would not be possible, or perhaps feasible, to engage in an activity—

[18] GPL 3 does make clear that it is a violation of the license, under Section 10.

such as practicing a standard or using a piece of software. In other words, if you can feasibly use the software without practicing the invention described in the claim, the claim is not necessary and therefore not licensed. Imagine, for instance, that a patent claim covers purple user interfaces and that a piece of software allows you to build interfaces in many colors. In this case, the claim covering the purple interface would not be a necessary one.

To assess the breadth of the capture, one must also understand who is considered an owner. For example, the grant may capture patents owned by a company and all its subsidiaries or affiliates, or it may capture patents owned by only a single entity. Capture provisions that include affiliates are intended to avoid licensors' creating loopholes by setting up separate companies to own patents—something that is often done for tax and other reasons. These entities are sometimes referred to as intellectual property holding companies. Most open source licenses, however, only capture the patents of one legal entity. A capture that includes parent companies can be a poison pill in an acquisition—a company may want to think twice about encumbering all its patents merely by buying another company that has contributed to open source software.

Also, the capture sometimes only includes patents owned by the licensor and sometimes captures all patents licensable by that entity. For instance, the definition of "Patent Claims in Mozilla Public License 1.1" is this: "Any patent claim(s), now owned or hereafter acquired ... in any patent Licensable by grantor." This could include patents not owned by the grantor but for which the grantor has the right to sublicense.

An open source license patent grant has only one field limitation, which is that the right is granted only in connection with the exercise of the copyright licenses granted for the software. So, if a contributor to an open source project grants rights in patents, those rights only extend to use of the software under the open source license and not to separate embodiments of the invention. Any other field limitations—such as territory, commercial or technology fields, or commercial vs.

noncommercial use—conflict with the Open Source Definition and therefore are never included.[19]

The time capture for an open source license patent grant is usually infinite and forward looking. Therefore, if the license captures any patents owned by the licensor, it will encumber any patent owned on the date of the grant and on any date in the future, such as if a patent is later prosecuted or acquired by the grantor. This is usually a concern for companies who expect to acquire other companies or technology; as soon as the acquisition closes, a patent license may spring into being. Most traditional patent licenses treat this subject in great detail, but in open source licensing, the capture is treated in a broad and shorthand manner.

To understand how patent licensing works in open source licenses, we will look at the patent provisions of Apache 2.0, which are both simple and customary for the open source context. Here are the relevant provisions from Section 3:

> Grant of Patent License. ... Each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable (except as stated in this section) patent license to make, have made, use, offer to sell, sell, import, and otherwise transfer the Work, where such license applies only to those patent claims licensable by such Contributor that are necessarily infringed by their Contribution(s) alone or by combination of their Contribution(s) with the Work to which such Contribution(s) was submitted. If You institute patent litigation against any entity (including a cross-claim or counterclaim in a lawsuit) alleging that the Work or a Contribution incorporated within the Work constitutes direct or contributory patent infringement, then any patent licenses granted to You under this License for that Work shall terminate as of the date such litigation is filed.

This provision has two parts: a patent license and a defensive termination provision. The patent license runs from every Contributor (i.e., the author of a contribution) and captures patents "necessarily infringed" by the Contribution.

[19] http://opensource.org/osd-annotated. See items 5, 6, 8, and 10.

In this paradigm, mere redistributors who do not contribute to the code base do not grant any patent rights. Therefore, a company that is using open source software and not redistributing, or that is redistributing but not redistributing any of its own modifications, need not wonder whether it is granting any rights.

In GPL 3, the scope of the patent grant is broader. Although its patent license, too, only must be granted by contributors, it covers all patent claims infringed by the software contributed to—not merely the patent holder's own code contribution.

Remember that the purpose of the patent grants in open source licenses is to make the software free from "submarine" claims introduced by contributors. The patent grants cannot, and are not intended to, manage the risk of claims by third-party non-contributors.

Patent grants in open source licenses are narrowly drawn to not extend to downstream modifications. So if you contribute to version X.1 and pass it along to me, and if I create version X.2, then I do not get the benefit of a patent license for any of my own contributions. Only upstream licensors grant rights in an open source license.

Defensive Termination

In a peripheral way, use of the code or mere redistribution does affect one's patents. If a party ("You") is exercising the license, and if it brings a claim accusing the open source software being provided under the license, then it loses the benefit of the patent licenses granted to it, but it does not lose not the copyright license.

Different open source licenses implement variations on this theme. For example, the Apache defensive termination provision is triggered by a defensive patent counterclaim. If that sounds harsh, keep in mind that the claim must accuse the Apache software in order to trigger the termination. Mozilla Public License has a broader defensive termination provision that also terminates the copyright license.

Following is a table of the patent license grants and defensive termination provisions of some of the most common open source licenses.[20]

Open Source Licenses with Patent Grants

License: Apache 2.0

Termination Trigger	Rights Terminated	Comments
3. (Grant of Patent License.) If You institute patent litigation against any entity (including a cross-claim or counterclaim in a lawsuit) alleging that the Work or a Contribution incorporated within the Work constitutes direct or contributory patent infringement.	3. Any patent licenses granted to You under this License for that Work shall terminate as of the date such litigation is filed.	The approach of Apache 2.0 is now probably the most common approach in open source licensing. Applies to claims accusing the work, and only patent licenses terminate.

[20] Copyright 2008–2017 Heather Meeker, heathermeeker.com.

Microsoft Reciprocal License (MS-RL)

Termination Trigger	Rights Terminated	Comments
3(C). (Conditions and Limitations) If you bring a patent claim against any contributor over patents that you claim are infringed by the software.	3(C). Your patent license from such contributor to the software ends automatically.	This approach is similar to one of the termination provisions of MPL 1.1 in that it terminates licenses from a particular contributor.

Mozilla 1.1

Termination Trigger	Rights Terminated	Comments
8.2. You initiate a patent litigation infringement claim (excluding declaratory judgment actions) against Participant alleging: (a) Contributor Version infringes any patent or (b) any software, hardware, or device, other than such Participant's Contributor Version, directly or indirectly infringes any patent.	8.2(a): All rights granted by Participant to You under copyright or patent licenses terminate prospectively. 8.2(b): Any rights granted to You by Participant under patent grants are revoked effective retroactively.	Only license with retroactive termination. Broad patent peace provision terminates all rights if a claim is brought related to the project.

Mozilla 2.0

Termination Trigger	Rights Terminated	Comments
5.2. If You initiate litigation against any entity by asserting a patent infringement claim (excluding declaratory judgment actions, counter-claims, and cross-claims) alleging that a Contributor Version directly or indirectly infringes any patent.	5.2. The rights granted to You by any and all Contributors for the Covered Software under Section 2.1 of this License shall terminate.	Unlike in MPL 1.1, termination is not retroactive and only extends to claims accusing the licensed work. However, copyright license also terminates, as in MPL 1.1. Similar to CDDL.

CDDL

Termination Trigger	Rights Terminated	Comments
6.2. If You assert a patent infringement claim (excluding declaratory judgment actions) against Initial Developer or a Contributor alleging that entity's contributed software directly or indirectly infringes any patent.	6.2 All rights granted to You by all Contributors terminate prospectively.	Broad patent peace provision terminates all rights, copyright and patent, but only for claims relating to the project.

CPL

Termination Trigger	Rights Terminated	Comments
7. If Recipient institutes patent litigation against a Contributor with respect to a patent applicable to any software (including a cross-claim or counterclaim in a lawsuit). If Recipient institutes patent litigation against any entity (including a cross-claim or counterclaim in a lawsuit) alleging that the Program itself (excluding combinations of the Program with other software or hardware) infringes such Recipient's patents.	Any patent licenses granted by that Contributor to such Recipient terminate prospectively. Recipient's rights under patent license from any Contributor terminate prospectively.	Patent peace provision terminates patent rights only.

Eclipse Public License (version 1.0 and 2.0)

Termination Trigger	Rights Terminated	Comments
7. If Recipient institutes patent litigation against any entity (including a cross-claim or counterclaim in a lawsuit) alleging that the Program itself (excluding combinations of the Program with other software or hardware) infringes such Recipient's patent(s).	Such Recipient's patent rights granted under Section 2(b) terminate prospectively.	Same as CPL except narrower patent peace provision.

GPL 3

Termination Trigger	Rights Terminated	Comments
10. You may not impose any further restrictions on the exercise of the rights granted or affirmed under this License. For example, you may initiate litigation (including a cross-claim or counterclaim in a lawsuit) alleging that any patent claim is infringed by making, using, selling, offering for sale, or importing the Program or any portion of it.	All rights can be terminated. (See general termination provision in 8.)	Note that this is structured differently from the other licenses. Applies to all recipients ("you") regardless of whether "you" are a contributor.

Chapter 14

Open Source and Patent Litigation Strategy[1]

When Richard Stallman wrote in GPL 2, "Any free program is threatened constantly by software patents," he crystallized the ideological battle between open source software and the software patent business. In 1991, when GPL 2 was released, that battle was in its nascent stages. Today, both open source licensing and software patenting have come into full flower, though their growth has proceeded on orthogonal axes; most open source software is never accused of patent infringement, and most software patent infringement suits don't accuse open source software. In fact, they so seldom directly interact that the lawyers who practice in these areas do not interact much either. This means those defending patent infringement suits may not be thinking about the tactics open source patent licensing offers to patent defendants.

In patent litigation defense, every little bit helps. Today, patent defendants should be paying attention to open source licensing and its possible effect on patent infringement claims. When you are sued for patent infringement by anyone other than a pure nonpracticing entity, your first areas of internal investigation should include the open source position of the plaintiff and, if you are considering retaliatory patent claims, your own open source position as well.

Patent lawyers may be surprised to know that while today most companies use open source software, most of them also struggle greatly

[1] This chapter appeared as "Open Source—The Last Patent Defense?" February 11, 2014, at outercurve.org.

with implementing the internal controls to coordinate their use of open source software with their patent portfolio management. This means it may be quite possible that a company will seek patent protection or seek to enforce patents that read on open source software the company is using or developing—a combination of activities that would often not be considered economically rational.

There have been at least two cases in which defendants have successfully used open source license enforcement as a defensive tactic in a patent lawsuit. The first case is the one most often cited to support the enforceability of open source licenses; most people forget that the case started as a patent claim. In *Jacobsen v. Katzer*, both parties developed and distributed software for controlling model railroads—with Jacobsen making his JMRI software available under an open source license free of charge and Katzer (via his company, Kamind Associates) selling commercial products under proprietary licenses. Jacobsen received a letter inviting him to license patents owned by Kamind, suggesting that the patents were infringed by the JMRI software. Jacobsen filed a declaratory judgment action asking the court to rule that the patents were invalid due to prior art (or failure to disclose prior art, including that of Jacobsen himself) or not infringed. As the patent case progressed, however, Jacobsen discovered that Katzer had copied some of Jacobsen's open source software and used it in Katzer's proprietary product, without the proper attributions and license notices. *Jacobsen v. Katzer* was finally settled in 2010, but only after becoming the seminal US case on open source licensing—not patent infringement—and resulting in a settlement payment by Katzer for violation of the open source license.

In *Twin Peaks Software Inc. v. Red Hat, Inc.,* Twin Peaks Software (TPS), which made proprietary network backup software, sued Red Hat and Red Hat's recently acquired subsidiary, Gluster. TPS claimed that the GlusterFS software—a network file system that aggregated multiple storage shares into a single volume—violated TPS's patent covering TPS's Mirror File System (MFS). Red Hat initially responded to the patent infringement suit by denying the infringement and asserting that the patent was invalid, but it later brought a

counterclaim alleging the TPS products incorporated open source software from Red Hat's product without complying with GPL. Red Hat sought an injunction against the TPS products. The case soon ended in a settlement, suggesting that TPS thought better of pursuing its patent claims in light of the facts.

In both these cases, the patent plaintiff was using open source software of the defendant and the patent defendant discovered a violation of the applicable open source license that it used to turn the tables on the plaintiff. In this way, open source license enforcement can be a substitute for a more traditional retaliatory patent claim. In each case, the plaintiff and defendant were in similar product markets—a very common context for patent litigation—which made the use of the defendant's open source code by the plaintiff likely. The moral of this story, for a patent plaintiff, is that one should have a robust open source compliance program in place before asserting a patent in a related space.

There are other, more subtle tactics as well. Open source licenses—particularly those written in the last 20 years—contain two kinds of provisions that bear upon patent litigation strategy. The first and more straightforward provision is the patent license. See for instance the license in Apache 2.0, which says this in Section 3:

> **Grant of Patent License.** Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable (except as stated in this section) patent license to make, have made, use, offer to sell, sell, import, and otherwise transfer the Work, where such license applies only to those patent claims licensable by such Contributor that are necessarily infringed by their Contribution(s) alone or by combination of their Contribution(s) with the Work to which such Contribution(s) was submitted.

This license only applies to contributors, so a mere re-user or redistributor of the software does not grant any rights. However, if a company has contributed to the software under an open source license or under a similar contribution license, then that company may have granted a license that can be used as a defense to an infringement claim.

For example, suppose company P (patent plaintiff) sues company D (defendant) for patent infringement. However, company P has

contributed software embodying the claims of the asserted patent to a project covered by this license. The Apache 2.0 license is a permissive license, so it may be easy for D to claim it is using software under this license. Raising this as a license defense can avoid liability—or at least create an unexpected defense that will add significant cost to P's prosecution of the suit.

Now consider the defensive termination provision of Apache 2.0:

> If You institute patent litigation against any entity (including a cross-claim or counterclaim in a lawsuit) alleging that the Work or a Contribution incorporated within the Work constitutes direct or contributory patent infringement, then any patent licenses granted to You under this License for that Work shall terminate as of the date such litigation is filed.

This means that by filing the lawsuit, P may have given up any patent licenses it has received from any contributors to the software—which may include D or third parties who may be aligned with D. This provision applies to all licensees, not just contributors. Even if D is not a contributor with patent claims to bring, bringing the claim exposes P to potential liability. Pointing this out may shift the balance in favor of the defense.

It's important to understand that patent defensive termination provisions in different open source licenses have different terms. Some, like Apache 2.0, are triggered by defensive claims, but some are not. Some, like MPL (or the corresponding "liberty or death" provision in GPL 3), also trigger a termination of the copyright license, making them even more powerful defense tools.

So the next time you are sued for patent infringement, you have not done your homework until you know the answers to all of the following questions:

- Is the plaintiff using any open source software of yours (related to the patent or not) in violation of the license?
- Does the asserted claim read on any open source software you are using? If so, would the complaint trigger a defensive termination provision that might apply to the plaintiff?

- Did the plaintiff contribute to any open source project any code under terms that would include a patent license? If so, do you have a defense under that license?

Investigating the last question can be an informational challenge, but it may not be as difficult as you think. Records regarding contributions may be available publicly, or open source projects may be willing to cooperate if it helps them defeat patent claims accusing their code.

The drafters of open source licenses intended to use the terms of those licenses to win a war against software patents, and whether they can do that remains to be seen, but in the meantime, patent litigators should not pass up the opportunity to use the principles of open source licensing to win their own battles as well.

Chapter 15

Trademarks

Open source licenses are copyright and patent licenses, and they say little or nothing about trademarks. Yet trademark is a robust and ubiquitous intellectual property right. Trademarks are cheaper and easier to enforce than patents, trademark law is clearer and better settled than copyright law (at least with respect to software), and unlike either copyright or patent rights, trademarks can last forever. The machinery for enforcing trademarks exists all over the world, and lawsuits for trademark infringement are very common and can yield significant damages. Overall, trademark is one of the strongest tools in a company's intellectual property arsenal.

But trademark law is quite different from patent or copyright law—so different, in fact, that some think it should not be called intellectual property law at all. At least theoretically, trademark law serves primarily to protect the consuming public, rather than the owner of the trademark. Most people think of trademarks as logos, like the Nike swoosh or the McDonald's golden arches. But in fact, a trademark need not be a picture or even stylized writing. Trademarks started as simply the name or sign a tradesman put on his product. A trademark can consist of just a company name, with no logo—whenever is used to designate the origin of a product.

The United States has a different trademark law from those of other countries: it is a first-to-use system rather than a first-to-file system. One gains rights in a trademark by using it in commerce, and one gains rights under the federal trademark laws by using the mark in interstate commerce. In some other countries, filing a trademark registration can make one the owner of a mark even without using it in commerce. So, trademarks are not like copyrights, which arise

immediately upon the work's fixation in the tangible medium. Trademarks are also not like patents in that registration is merely evidence of ownership and not ownership itself.

A trademark represents the source of origin of a product, and a service mark represents the source or origin of a service. For purposes of this discussion, we will only discuss trademarks, but service marks work in roughly the same way. Because it represents a source, a trademark also represents the quality of goods associated with the source. If you buy a coffee at a Starbucks coffeehouse in a strange city, you assume the product you buy will be like the coffee you get in a Starbucks coffeehouse in your hometown. If just anyone could use that trademark, you might be fooled into buying something of lower quality or something entirely different from what you expect. Using trademark law, a court can prevent counterfeiters from misusing the trademark so the public will not be misled.

Trademark law imposes more duties on the owner of the intellectual property than does patent or copyright law. It is relatively difficult to lose the copyright in a work, other than by expressly dedicating the work to the public domain. It is possible to lose the rights in a patent by not paying maintenance fees to the Patent and Trademark Office (PTO). But trademark law requires that the owner police a trademark to maintain rights in it. If it were otherwise, the trademark would not protect consumers. A trademark owner is expected to maintain quality control over goods that bear the mark, including any goods that are made by others under license. Trademark law is therefore a "use it or lose it" regime. When an owner of a trademark fails to do this, it is called *dilution*, *genericness*, or *blurring*. Some very famous marks—like aspirin and heroin—have become generic and no longer point to any particular source.

Trademarks in the Open Source World

Therefore, open source software licensing and trademarks make strange bedfellows. They are inherently opposed schemes; while open

source licenses allow anyone to modify code,[1] no trademark owner can allow use of the trademark on that code without controlling the nature of the resulting product.

Early open source licenses dealt obliquely with this issue. For example, early forms of BSD contained this:

> Neither the name of the University nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

In substance, this clause states that no trademark license is granted to use the name of the University (of California). Other licenses contain variations of such language. Although their drafting is awkward, they probably mean to say that no express trademark license has been granted.

Many of the more sophisticated open source projects recognize the potential conflicts between open source licensing and trademark principles. They also recognize that the average open source developer is probably unaware of what is or is not allowed under trademark law. Therefore, many open source projects set written guidelines for trademark use. These trademark guidelines usually describe what is not allowed, what is always allowed, and what is allowed with a license. Most of their content simply maps existing trademark law onto the use of the project's name. [2]

Attorneys who are accustomed to managing trademarks in conventional commerce might find these policies surprising; the conventional way to manage a brand is to allow no use at all, except pursuant to a formal license agreement, and to be aggressive about enforcement. But in the open source world, aggressive enforcement practices, particularly against hapless infringers, can have the effect of alienating the very community the company is trying to promote. So open source projects often allow use of the trademark for purposes such

[1] See the open source definition at www.opensource.org/docs/definition.php.

[2] For an example of a model policy, see http://modeltrademarkguidelines.org/index.php?title=Home:_Model_Trademark_Guidelines.

as naming user or developer groups, holding conferences, and creating T-shirts and other promotional items. These are all actions that would not be allowed in conventional brand management.

What's in a Name?

As a result, open source software licensing is a confused sea of brand management. That is not surprising, nor is it necessarily a problem because it may be unclear whether the names of projects are intended to be used as trademarks at all. Many of them probably are not; every project needs a name, but not all of them are whole products and many of them are only intended as templates for skilled developers to create products. Therefore, such projects might not enjoy the quality control that a finished product might warrant. (For lawyers, think about the forms and models that you get from your colleagues and that are usually provided with the admonition that they need to be adapted for your use.) However, it seems likely that branding fights will become more common in the open source world as open source software becomes more integral to commercial products.[3]

As a case in point, consider the brand management, or lack thereof, for the most popular and valuable piece of open source software—the Linux kernel. Many people associate Linux with a bowling pin–shaped penguin logo. That penguin, nicknamed Tux, is not actually a trademark.[4] The copyright in the original logo is dedicated to the public domain, and variations of it have been used for many Linux distributions. It has sometimes been called a mascot rather than a trademark (and it appears on the cover of this book).

The name *Linux* is based on UNIX and the name of the original author of the kernel, Linus Torvalds. The use of Linux as a trademark is policed—at least nominally—by the Linux Mark Institute

[3] For an excellent article on genericness in open source trademarks, see Pamela S. Chestek, 2013, "Who Owns the Project Name?" *International Free and Open Source Software Law Review*, 5(2), 105–120. See www.ifosslr.org/ifosslr/article/view/87/159.

[4] The use of the penguin has provided fodder for plenty of amusement. See the discussion of the meaning and history of Tux at http://en.wikipedia.org/wiki/Tux

(www.linuxmark.org). However, at this point, the name *Linux* has been used so broadly and on so many different products that as a trademark, it may be generic and unenforceable.[5] Nevertheless, if a company were to try to sell a Linux product that bore no resemblance to the Linux kernel or that was not licensed under GPL, then even if there were no trademark claims to be made, such a use might give rise to consumer lawsuits claiming the name was misleading. Given the tendency of businesses to try to capitalize on the label *open source* in many areas, such claims seem inevitable.

The moral of this story is that while best practices for brand management for a company running an open source project are fundamentally no different from those for any other product, they may need to be adjusted to suit the business objectives of the project. It's best to clearly distinguish from your other product lines any open source project you may run. This will help you avoid having the brand on your other products become generic. Also, setting branding policy is partly an educational effort, helping the community understand how to avoid infringing the company's trademarks.

But the most important part of trademark management in open source is to think about it in the first place. What you call your project may become very important, and you and your community may very well have an interest in avoiding a war of versions that can confuse potential users and developers.

[5] Confusion over what constitutes a "Linux" product is evidenced by the FSF's strophic explanation of the difference between Linux and GNU/Linux. The GNU code base includes a set of tools promulgated by the GNU Project. These tools are usually part of a product distribution that contains the Linux kernel. See Wikipedia's definition of Linux: "Strictly, the name Linux refers only to the Linux kernel, but it is commonly used to describe entire Unix-like operating systems (also known as GNU/Linux) that are based on the Linux kernel and libraries and tools from the GNU project." See en.wikipedia.org/wiki/Linux.

Part V

Contributions and Code Releases

Chapter 16

Open Source Releases

This chapter is about releasing new software—software a business has written for itself—under an open source software license.

Maybe your company has not yet arrived at the point of wanting to do this. But if you keep using open source software long enough, you will eventually see the value of giving back. Free-riding on the work of others may seem like manna from heaven, but cooperative development is usually even more rewarding. Even profit-making companies may decide to release open source code, for many reasons. For example, they may want to promote the adoption of a certain protocol or method, and making source code available may be the quickest and easiest way to accomplish that goal. Or, they may decide that an existing product is stale, commoditized, or no longer profitable and wide adoption of it might benefit their other business lines.

Sometimes pressure to release code as open source comes from the company's engineering staff. This can create competing interests. In the short term, it's not a responsible use of shareholders' capital to develop software only to give it away. Sometimes, engineers consider releasing open source software to be a résumé enhancement, and they may put the interest of their own reputation ahead of their company's bottom line. But many times, the competing interests align when giving away software is good for business.

When companies decide to release open source software they have written, they often ask, "What license should we use?" In a way, the choice of an open source license is a choice between many options, each of which is imperfect in some way. Some companies then assume they should write their own license, and this is generally not a good idea. A license often backfires, and even if it is clearer and better drafted than

the common alternatives, introducing a unique license will probably substantially hamper the very adoption that the company is trying to achieve with its code release.

Figure 16.1 represents one way to approach selecting a license for an open source code release. Note that realistically, the licenses referenced in the diagram are the only good choices for open source releases. They are all very common and well understood.

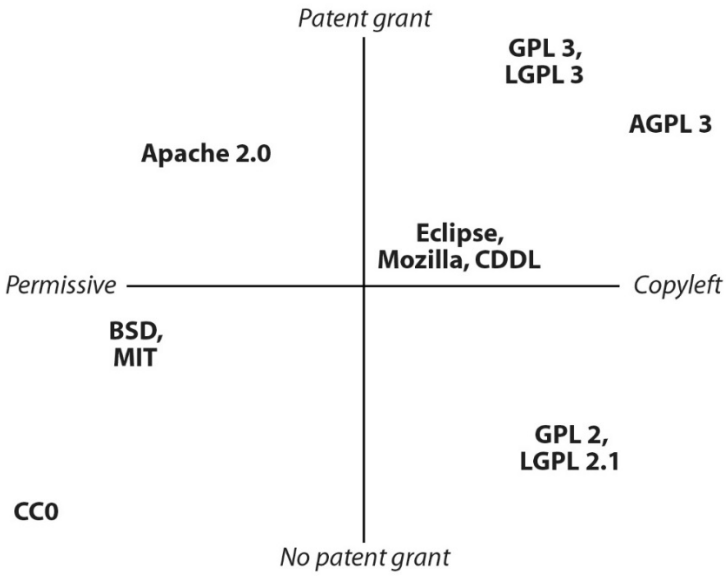


Figure 16.1 Dimensions of license selection

There are two axes in the diagram, each of which represents a decision. The X (horizontal) axis is copyleft. Here, there is a spectrum of choices. However, the most important factor is your answer to the question, "Do I want to use a copyleft or a permissive license?" Most permissive licenses are fungible from a copyright point of view, meaning you could use any of them. Also, CC0 is actually a dedication to the public domain, rather than a license. The difference between CC0 and BSD or MIT is that CC0 does not require any license notice to be retained; CC0 waives all copyrights (but not patent rights) in the code.

The Y (vertical) axis represents your yes-or-no answer to the question, "Do I prefer an express patent grant in the license?" Generally, licensors who have valuable patent portfolios to manage prefer licenses with express grants. This is because such licensors are concerned about the scope of implied grants.[1]

If you decide on copyleft, you need to choose how strong a copyleft regime you want—from AGPL (the strongest) to Mozilla or CDDL (the weakest). Here, *strong* or *weak* refers to how much the license attempts to govern the delivery of source code for other code that is integrated with the open source code. (Refer back to Chapter 3: Common Open Source Licenses for details.)

Once you have made these two choices, the license nearly selects itself. More importantly, "Which license?" is not really the right question. The right question is, "What are my goals for releasing this code, and how does releasing it help my business?" Below are some use cases that might help illustrate how to answer that question. Of course, it's fine to release any software under a permissive license if you want to maximize adoption and don't care about others free-riding on your code. But when it comes to copyleft licenses, some of them don't make sense for certain use cases.

- **Development utility**. This kind of program is not usually distributed—it is usually used internally. Therefore, strong copyleft licenses like GPL are fine, but they won't do much to encourage sharing of source code. The source code disclosure conditions of GPL only attach on distribution. Even a license like Affero GPL will probably be of little help in this regard.
- **Content management system**. This kind of application is a good choice for AGPL. Where it is not used internally, it will

[1] The law on implied patent licensing is vague—see Chapter 13: Open Source and Patents. However, there is also a school of thought—less popular—that using a license with no express grant means no patent rights have been granted. Some companies also choose to release code under a license with no patent grant—like BSD—and then grant a separate patent license. (See, for example, Google's grant of additional rights for WebM at www.webmproject.org/license/additional/).

probably be used to provide services, and AGPL will require users to share source code. GPL will not.

- **User application**. This kind of program often gets distributed, and because it is a stand-alone program, GPL makes sense.
- **Code library**. LGPL or one of the weak copyleft licenses is a good choice in this case. If you choose GPL instead, you will reduce adoption because using the library in an application will require the developer to put the entire application under GPL.
- **Non-software**. Don't confuse the world by releasing things like documentation, photos, music, or books under software licenses. Use a Creative Commons license instead. Conversely, don't release software under non-software licenses like CC-BY-SA. It's a good idea to dedicate code with limited copyright protection to the public domain. Don't make the world try to figure out the extent of copyright law to use your release.

Here are some examples of business cases and how they might map onto these license choices.

Helper Code: CC0

A company sells a wearable personal note-taking device that is wireless-enabled and can interface with HTML applications. A small reference code routine can be embedded in the application to implement the device's API. The code is very simple, probably has little copyright protection, and must be built directly into the application. This company chooses CCo because it wants to increase adoption and does not want to require its developers to provide any attribution or notices on a small device.

Adoption of Open Standards: Apache 2.0

A company has created a new local area communications protocol. The company wants as many developers and users as possible to start using the new protocol. To facilitate this, it releases some development libraries that can be included in applications to implement the standard. This code offers the ability to encode and decode into standard

compliant formats. The company has not sought to protect this standard via patent protection. Instead, its business model seeks to encourage as much widespread adoption as possible while selling applications and products that meet the standard. In other words, this is an open standard. Also, the company would prefer to have a licensing notice on the code to publicize the availability of the code and its company name. The company chooses Apache 2.0 because it is a permissive license that will give developers and users confidence that the company is not seeking to reserve patent rights.

Dual Licensing

The company's business model seeks to provide an application for personal financial management that can be used to provide online services. The company plans to make a stripped-down version of the software available under an open source license; it also wants to offer a fuller-featured version—which will have the ability to support multiple mirrored servers—for a license fee. The company chooses Affero GPL 3. It also offers an alternative license under proprietary terms.

Disrupt Competition

The company has been selling, for many years, a spell-checking utility. But recently, the company's sales have dwindled because there are so many competitive products on the market. Maintaining the product has become expensive, and one of the company's competitors still has an edge in sales. However, if the competitor's product becomes a standard, the company will have to engage in significant engineering efforts to make its product compatible with that standard. The company chooses GPL 2. By doing so, it creates a free substitute for its competitor's product while preventing the competitor from privatizing the code in its own products (because the terms of GPL don't allow that).

Trademark Stewardship

Whenever a company launches a new product, it needs to consider branding and trademarks. Trademarks are among the most important elements of the intellectual property used to insulate a product from competition. Releasing an open source project is no different, but branding in the open source world can be tricky.

There is an inherent tension between open source licensing and trademark licensing. However, the two can coexist if properly stewarded. (For more on trademark law and open source in general, see Chapter 15: Trademarks.) This section focuses on the business strategy for doing a code release and best practices to steward trademark rights in the context of an open source code release. Trademarks protect not only their owners but the open source community as well. Developing conscientious trademark practices will help not only the company running the project but the consumers and developers using the code as well.

Be Careful How You Mix Marks

Companies that release software under open source licenses should always ensure that the open source software is under a brand that is distinct from their other products. It can be tricky to maintain strong trademark rights in open source code, so if something goes wrong and the rights weaken, the company's other products should not be compromised. For example, lots of companies like to call products "FOOBAR" and "Open FOOBAR" (or "FOOBAR Community Edition"). This works well for dual-licensing initiatives, but not for independent open source projects that will be primarily under community (rather than company) control. Companies releasing software for the purpose of starting a community project should choose a new brand for the project. When the project is no longer under company control, the trademark will no longer represent the company's quality control and will therefore no longer be, strictly speaking, an appropriate product to bear a company mark.

Transparent Policies

A code release should always include a coherent and transparent trademark policy. Open source licensees rarely understand the distinction between open source licensing and trademark use. Having a policy not only helps the company establish systematic quality control to maintain the rights in the mark, but it helps the user base avoid accidentally infringing those rights. Traditional trademark policies allow users to do almost nothing; open source policies need to be more moderated and walk a fine line between control and freedom. For an example of a model policy, see

modeltrademarkguidelines.org/index.php?title=Home:_Model_Trademark_Guidelines.

Here are some things to consider when developing such a policy:

- If a license modifies the code, can it still retain the mark? Most policies say that any modification requires removal of the mark, but some allow minor changes like configuration or default settings to be changed.
- What uses of the project name can be made by user groups or developer groups? A trademark policy will usually require express permission to register domain names that include the project mark or any variation of it. Some policies allow user and developer groups to create merchandise to help raise funds; some do not.
- To avoid mistakes, it's best to remove logos and trademarks from the public source tree. While some companies may make binaries available for download with the logos intact, preserving the logos from the source tree may inadvertently cause misuse of the trademark by licensees.

Contribution Agreements[2]

Once you have decided what license will govern the rights granted by your project to the world, you should consider what will govern the rights granted by the world to your project. Many open source projects use inbound licenses called *contribution agreements*. The objective of a contribution agreement is to make sure the project has the latitude to change the outbound license it applies to the project. For a project made available under a permissive license, the question of whether to use a contribution agreement is not as crucial or complicated, so here we will only focus on the case of a project under a copyleft license.

Contribution agreements have many variations, but the most common is the Apache Corporate Contribution License Agreement (CLA).[3] Other variations on contribution licenses include assignments rather than licenses and restrictions on the relicensing of code (such as limiting it to open source licenses). However, these variations are less common than a simple unrestricted license—roughly the equivalent of a permissive open source license, but without the notice requirements. Contribution agreements are not standardized, except to the extent that the Apache CLA is a de facto standard. A project can make whatever bargain with its contributors that it likes. Some projects will accept contributions under different terms from different contributors, and some projects are more transparent about this decision than others.

Of course, the simplest and most transparent way to accept contributions is not to use a CLA at all. This approach is popular among some in the open source community, but it can hamstring and even kill open source projects that decide to change their license terms. If you accept contributions under GPL, for instance, the project can never change to LGPL or Apache without a potentially cumbersome or

[2] An earlier version of this section appeared in "Legal Issues: The Contribution Conundrum," *Enterprise Open Source Journal*, July 2006. There are also more comments here: http://heathermeeker.com/2014/06/30/contributions-and-the-latest-generation/.

[3] www.apache.org/licenses/cla-corporate.txt.

impossible relicensing effort.[4] Projects that do not use contribution licenses sometimes use a certificate of originality, which does not dispose of licensing rights[5] but which requires the contributor to show that the contribution is his original work—therefore providing evidence that contributing it is not an infringement of third-party copyrights.

The Free Software Foundation generally requires that all rights to contributions to FSF projects be assigned to it.[6] The theory is that this approach makes it easier to enforce rights in the outbound open source license because the project owns all the code and therefore has standing to sue for copyright infringement for every portion of the code base. However, most projects do not use this approach—relying instead on a copyright in most of the code, as well as on ownership of the copyright (however thin) of the compilation of all of the various contributions.

Those who object to contribution agreements mainly want the project's practices to be transparent—the mantra being "License in equals license out." They prioritize the contributor's right to know the disposition of his contribution over the freedom of the project to change its licensing policies.[7]

Relicensing Code

If a project decides to change its outbound license, it almost always changes from a more restrictive license (like GPL) to a less restrictive one (like BSD). This is because the reverse does not make much sense; once code has been released under an open source license, the rights can

[4] It is generally understood, for instance, that changing the license for the Linux kernel from GPL 2 only to anything else is impossible, given the number of contributors and lack of a contribution agreement.

[5] See for example the Developer Certificate of Origin, http://elinux.org/Developer_Certificate_Of_Origin.

[6] www.gnu.org/licenses/why-assign.html.

[7] For a good exposition of the view against contribution agreements, see Simon Phipps, "Governance for the GitHub generation," *Infoworld*, June 30, 2014, www.infoworld.com/article/2608195/open-source-software/governance-for-the-github-generation.html.

never be taken back. The project can release the code under a more restrictive license, but the rights granted under the less restrictive license will still be available. Occasionally, relicensing under a more restrictive license can be effectively done with a major update; in that case, the old version will be available under the old license, and the new elements will be limited to the more restrictive license.

Of course, the ability to relicense may depend on whether the project has used a contribution agreement. If, for instance, a project starts under GPL and does not use a contribution agreement, all contributions will be covered by GPL and cannot be changed without the contributor's permission.

Therefore, if a project is unsure about which license to use, it is best to start with a more restrictive license and move to a more permissive one if it becomes evident that more permissiveness will serve the project better.

Corporate Organization

Some companies take the extra step of creating a separate entity to run a new open source project. Doing so can be a good strategy for untangling intellectual property rights in the open source project from the rest of the company's operations. Also, because large open source projects can involve market competitors, the separate entity gives the participants a level and transparent playing field on which to engage in interactions that might otherwise raise antitrust concerns. An entity specially created to run open source projects often takes the form of a not-for-profit entity. However, the last few years have seen many travails for US companies seeking not-for-profit status.[8]

8 See Ryan Paul, "IRS Policy that Targeted Political Groups Also Aimed at Open Source Projects," *Ars Technica*, July 2, 2014, http://arstechnica.com/tech-policy/2014/07-irs-policythat-targeted-tea-party-groups-also-aimed-at-open-source-projects/. For more on the tax decision, see Heather Meeker and Stephanie Petit, "The New Foundations of Open Source," 33 *Santa Clara High Technology Law Journal,* 103 (2017).

License Choice

License choice is less complicated than it may seem on its face, mainly because it is unwise to choose any open source license other than the most popular ones—BSD, MIT, Apache 2.0, GPL2 or 3, LGPL 2 or 3, AGPL3 or MPL 2.0. The rules below are generally embodied in a simple script available at heathermeeker.com/license-picker-2-0/. This approach is intended to address most cases in which a private business wants to release open source software.

First, eliminate the cases for which open source licenses don't work:

1. If you want to restrict the use of your software (such as to non-commercial use or limiting SaaS use), you do not want an open source license. Consider the PolyForm licenses.[9]
2. If you want maximum adoption and do not care about getting any attribution, consider dedicating your copyright to the public domain using CC0.

If you got this far without an answer, now you need to choose a permissive or copyleft open source software license.

3. If you want maximum adoption and want to require license notices, you want a PERMISSIVE license. You then have three choices: MIT, BSD and Apache 2.0.

3.1. If you want to grant express patent licenses, choose Apache 2.0.

3.2. If you want your license to be compatible with GPL2,[10] choose MIT or BSD. MIT and BSD are similar enough so that there is rarely any clear reason to choose one or the other.

4. If you want to discourage privatization of your code, you want a COPYLEFT license. You then have six choices:

[9] www.polyformproject.org.

[10] GPL2 is the license that applies to the Linux kernel, so the need for a GPL2-compatible license often drives license choice.

GPL2 or 3, LGPL 2 or 3, AGPL3 or MPL 2.0. The following four rules will help you with this choice, but no one license may satisfy all your requirements.

4.1. If you want a COPYLEFT license and want to grant express patent licenses, you must choose GPL3, LGPL3, AGPL3, or MPL 2.0.

4.2. If you want a COPYLEFT license and want your license to be compatible with GPL2, you must choose GPL2, LGPL2 or MPL2.0. (Note that only MPL2.0 satisfies the conditions in both 4.1 and 4.2.)

4.3. If your code is not a whole program, but a library, do not pick GPL or AGPL; LGPL or MPL is the correct license for libraries.

4.4. If your code will be used in IoT devices, do not choose any version 3 license. (Note that only MPL2.0 satisfies the conditions in both 4.1 and 4.4.)

4.5. If your code will primarily be used in SaaS, consider choosing AGPL 3.0.

Also, free software advocates will certainly take issue with the premise that the version 3 licenses—GPL 3 and LGPL 3—will limit adoption, or should not be used in IoT. At this time, many companies have internal policies against using code under these licenses, and almost none will use version 3 code in IoT. (For a more detailed discussion, see Chapter 10: GPL 3 and Affero GPL 3.)

Generally, permissive licenses will encourage adoption more than will copyleft licenses. Most businesses have few internal policy restrictions on using permissively licensed code. A business may want to choose copyleft for public relations and moral or political reasons, but most private businesses choose copyleft licenses for code releases in order to disrupt competition. Permissive licenses easily enable competitors to privatize code in their own products; copyleft licenses require competitors to make source code available, which substantially reduces any potential free-rider gain by a competitor—at least for distributed software.

Part VI

Additional Topics

Chapter 17

Mergers & Acquisitions and Other Transactions

The handling of open source issues in transactions has changed radically over the last two decades. Today, almost all software contains some open source elements, so it's clear to almost everyone that asking a vendor or seller to represent that it uses no open source software is archaic and pointless. Today, most transactions focus on disclosure and allocation of risk. One sea change that has occurred is that open source representations are now almost as common in day-to-day procurement or development deals as they are in mergers and acquisitions (M&A) and investments.

Nevertheless, open source representations are almost always poorly written—either too narrow or too broad (or sometimes both) to the point of being nonsensical. In some ways, open source representations are the same as those for other technology and intellectual property; in other ways, they are quite different.

First, consider this representation, which is very common in all sorts of technology transactions:

> The use of the Software will not infringe any third-party intellectual property rights.

If this warranty appears in your deal, you have already covered most of the risk attendant on the use of open source software. Violating open source license conditions causes the license to terminate (or be quickly subject to termination, in the case of the few licenses that allow cure periods for violation of license terms). Therefore, violating open source conditions will trigger a breach of this representation. However, many lawyers erroneously assume that specific additional open source

representations are needed to address liability. In fact, they are probably not necessary for that purpose, although additional information (sometimes called a *listing representation*) is useful for due diligence.

Most transactions today require a disclosure of basic open source information so the buyer can conduct its own diligence. Such disclosures in day-to-day transactions are usually limited to self-reporting. Most M&A transactions today use an outside code scanner to do a forensic analysis. (For some of the challenges of conducting due diligence on self-reported open source disclosures, see Chapter 12: Audits and Due Diligence.)

Although such information can certainly be requested separately as a matter of due diligence, it is usually tied to a listing representation, compelling the vendor or seller to disclose the open source software included in a product as part of an exhibit or disclosure schedule. Drafting and listing representations in such agreements can be more challenging. Many of them ask for too little information and too much information at the same time.

The first step in drafting open source provisions for agreements is to define *open source software*. Here is a sample definition that contains some of the drafting issues that are most common:

> *Public Software* is any software that is used by the Company and made generally publicly available in source code form or under any public source, freeware, community, or open source license approved by the Open Source Initiative, including the Sun Binary Code License, the GNU Public License, and the Netscape Public License.

The above example is fairly awful, with many defects:

- *Public Software* is not a term of art used by the open source community.
- Names licenses that have been deprecated.
- Names licenses that don't exist.
- Assumes all open source licenses are approved by OSI.
- Includes freeware, which is proprietary software.

- Includes all software that is always in source code form (like JavaScript).[1]
- Includes software owned and released by the Company, rather than licensed to the Company by third parties.
- Requires disclosure of all software used by the Company, instead of software in the products.

Here are some sample definitions that I think work better:

> "Open Source License" means any license meeting the Open Source Definition (as promulgated by the Open Source Initiative) or the Free Software Definition (as promulgated by the Free Software Foundation), or any substantially similar license, including but not limited to any license approved by the Open Source Initiative or any Creative Commons License. For avoidance of doubt, Open Source Licenses include without limitation Copyleft Licenses.
>
> "Copyleft License" means any license that requires, as a condition of use, modification or distribution of a work of authorship, that such work of authorship or derivative works thereof be made available free of charge under such license, and that, in the case of software, be made available in source code form, or under terms that allow such software to be reverse engineered. Copyleft licenses include without limitation the GNU General Public License, the GNU Lesser General Public License, the Mozilla Public License, the Common Development and Distribution License, the Eclipse Public License, and all Creative Commons "sharealike" licenses.

Creating a separate definition for *Copyleft License* can help one avoid requiring unnecessary disclosure for permissive open source software. The purpose of these representations is to compel the disclosure of enough information to conduct due diligence. However, they should not provide so much information as to flood the seller with make-work and the buyer with unnecessary information that will slow down the diligence process and, ultimately, result in disclosures that might shift excessive risk to the buyer.

> "Open Source and Copyleft Materials." All use and distribution of Company Products by Company, and all use by Company of any

[1] For more on this topic, see Chapter 2: A Tutorial on Computer Software.

> materials subject to an Open Source License, is in compliance with all Open Source Licenses applicable thereto, including without limitation all copyright notice and attribution requirements and all requirements to make available source code. Section [_____] of the Disclosure Schedule lists all Open Source Materials (including release number, if any) included the Company Products, and (1) the Open Source License (including version number, if any) under which Company uses such Open Source Materials, (2) the location on the Internet, if any, where the Open Source Materials were most recently accessed by Company, (3) whether the Open Source Materials have been modified by or for Company, (4) whether the Open Source Materials have been distributed by or for Company, (5) the applicable Company Product, and (6) with respect to Copyleft Materials, how such Copyleft Materials are integrated with or interact with the remainder of the applicable Company Product. Without limiting the foregoing, Company has not used Copyleft Materials in a manner that requires the Company Products, any portion thereof, to be subject to any Copyleft License.

Today, most attorneys understand that the open source software elements in a product are of far more concern than any used by the vendor or seller for back-end processing or even product development. Accordingly, the need to compel disclosure has been narrowed to the elements actually used in the product that forms the value basis for the transaction.

Allocation of Risk

Clients frequently ask me what is reasonable and customary in allocation of open source-related risk in transactions. They are either convinced that a vendor should bear no liability for third-party open source software or that it should bear all liability for it—usually depending on what side of the deal they are on.

Of course, there is no one answer to such a question. Allocation of risk in transactions usually depends on who has the greater bargaining power. Plenty of vendors have agreed to bear risk for open source software over which they have no control in order to get deals done. But there are rational ways to assess the issue.

Often, those negotiating deals find it hard to allocate risk because they view it as a moral issue. It is not—it is a commercial issue. There

are two overarching ways to approach allocation of risk: control and pricing.

One school of thought holds that risk should be borne by the party in a transaction who can best control it. This makes sense from a practical viewpoint, and it also appeals to the need for a moral element of blame for problems. According to this theory, the party that makes a decision to use third-party software should bear the risk. Of course, that is almost always the vendor.

Another school of thought holds that the party who is making the money on the transaction should bear the risk. In accounting terms, the party receiving payments for the product can reserve some portion of the purchase price against future liability or loss. Of course, this is almost always the vendor as well.

Although both of these theories are useful, they do not tell the whole story. Take, for example, a customer who buys a storage device that consists of off-the-shelf hardware, an application for the Linux platform, and a Linux system. Let's assume the application, which provides the most added value in the product, consists primarily of code written by the vendor and is proprietary. Who should bear the risk for the Linux system?

This can be a thorny question to answer. While it is true that the vendor chose the software, it may be unrealistic today to expect a vendor of such products to use anything else. Linux is very popular in embedded systems. Also, at least theoretically, the customer benefits from the royalty-free nature of the Linux platform and therefore pays a lower price. So even though the vendor is receiving payment for the product, the customer is saving money by choosing the open source–based product.

As another example, consider a product that consists of an e-commerce application that runs on a LAMP stack. The vendor may, for convenience, deliver a working instance of the software that includes both the application and the LAMP stack. But the customer could just as easily have procured the LAMP stack on its own, or the customer could have chosen an application that runs on a Windows system rather

than on a Linux system. In this sense, using Linux is as much the customer's decision as the vendor's.

Because of these nuances, the vendor often does not bear risk for third-party open source.

Keep in mind, as well, that there are different categories of risk—the two main ones being performance warranties and intellectual property warranties. *Performance warranties* say that the product will function as advertised, and *intellectual property warranties* say that the product will not infringe third-party intellectual property rights. Vendors almost always undertake risk for performance warranties. That is because they usually have confidence in the open source software they have selected, and their quality control checks are not very dependent on whether the software in the product is open source or not.

Intellectual property warranties are different. For the most part, the vendor has no intellectual property coverage from its own upstream supplier, which may be a noncommercial open source project or a commercial vendor such as Red Hat. Keep in mind that open source licenses are granted as-is, so there is no coverage except by separate contract.

However, there are sometimes nuances to how much liability a vendor undertakes for third-party open source components to its product. The vendor will, more often, undertake liability for libraries or other small open source components that are integrated into the vendor's product. But as open source computing platforms become more popular, the vendor may disclaim liability for infrastructure software such as the LAMP stack, Hadoop, or OpenStack.

Ironically, these popular infrastructure elements are probably the least risky when considering third-party infringement liability. Copyright and trade secret claims accusing open source projects are actually quite rare. The main risk is third-party patent infringement. And more ironically, this is the issue upon which open source and proprietary software differ the least—and upon which the vendor's own code and open source code differ the least. Any of these can be subject to third-party infringement claims, and even proprietary vendors try to

avoid liability for third-party patent claims on code they wrote themselves.

Ten or twenty years ago, it was common for purchasers in M&A deals to insist that open source representations were *special representations*—for which liability might be enhanced or even unlimited. In M&A deals, liability is usually quite limited, often to 10 to 15 percent of the purchase price, via an escrow against which the buyer can make claims for breach of representations and warranties. Today, open source representations are unlikely to be treated any differently from other intellectual property representations, and rightly so, because open source intellectual property problems are, generally speaking, no more risky than any other intellectual property problem. The clamor for enhanced remedies probably sprang from the mistaken idea that "viral" licenses caused proprietary code to be automatically subject to those licenses. If that were true, then enhanced remedies would be appropriate, but it has never happened and is not even likely to be possible under law.[2] Once the fear of this catastrophic result subsided, buyer insistence on unlimited remedies for breach of open source representations subsided as well.

Drafting the Customer Agreement

If you are responsible for preparing customer agreements for proprietary products that include open source software, there are a few rules you should follow.

The most important is that GPL and LGPL do not allow licensing of binaries on different terms. For this reason, it is actually a violation of the license to purport to relicense a product including these elements under other terms. If you plan to include GPL or LGPL components in your product (assuming your plan to do so is compliant—for which see Chapters 8 and 9 on GPL and LGPL compliance), you must carve out

[2] For a discussion of copyleft licenses and their remedies, see Chapter 6: What Is Distribution?

the licensing for those products from your customer agreement. This carve-out might look like this:

> Notwithstanding the foregoing [reference license grant], Licensee acknowledges that certain components of the Software may be covered by so-called open source software licenses (*Open Source Components*), which means any software licenses approved as open source licenses by the Open Source Initiative or any substantially similar licenses, including without limitation any license that, as a condition of distribution of the software licensed under such license, requires that the distributor make the software available in source code format. To the extent required by the licenses covering Open Source Components, the terms of such licenses will apply to such Open Source Components in lieu of the terms of this Agreement. To the extent the terms of the licenses applicable to Open Source Components prohibit any of the restrictions in this Agreement with respect to such Open Source Components, such restrictions will not apply to such Open Source Components. To the extent the terms of the licenses applicable to Open Source Components require Licensor to make an offer to provide source code or related information in connection with the Software, such offer is hereby made. Any request for source code or related information should be directed only to _____. Licensee acknowledges receipt of notices for the Open Source Components for the initial delivery of the Software.

Most of the weak copyleft licenses allow licensing of binaries under other licenses, as long as the source code is made available under the copyleft license. Because this is the simplest approach for a binary distribution, it is the approach most companies take. The provision above says, "To the extent required by the licenses covering Open Source Components, the terms of such licenses will apply to such Open Source Components in lieu of the terms of this Agreement." In such cases, it is not required. The same is true, of course, for permissively licensing open source components.

The provision above is purposely general. It is possible to "hard code" the exceptions into your license, but then you must revise it every time the open source components in the product change. Most companies find it easier to avoid such an approach, which could result in noncompliance.

One element of the provision above is specifically directed to LGPL components. LGPL 2.1 says this in Section 6:

> As an exception to the Sections above, you may also combine or link a "work that uses the Library" with the Library to produce a work containing portions of the Library, and distribute that work under terms of your choice, provided that the terms permit modification of the work for the customer's own use and reverse engineering for debugging such modifications.

End user licenses for proprietary products usually include reverse-engineering prohibitions that would conflict with this provision of LGPL.

Keep in mind that the carve-out language above prioritizes compliance with the open source license over potential conflicts in the proprietary EULA. This is based on the assumption that a minor incursion into the EULA's terms—such as making its reverse-engineering prohibitions inapplicable to reverse engineering for dubbing modifications of LGPL code—is a small price to pay for ensuring compliance with the open source license and avoiding losing the license to the open source software.

Development Agreements

Companies that engage developers to create software have long used agreements that assign intellectual property rights to the customer.[3] A typical development agreement might include the following buckets of intellectual property:

- Materials developed by the developer prior to or independent of the project. The materials are licensed to the customer, usually without restriction.
- Materials newly developed by the developer for the project. The intellectual property rights in these materials are usually assigned to the customer.

[3] These are sometimes referred to as *work for hire* agreements—though that is a misnomer. The work made for hire doctrine of copyright law generally applies only to employees.

Today, however, a great deal of development either consists of modifications to third-party open source software or requires the integration of third-party open source software into the material being developed. This new development paradigm requires a more nuanced intellectual property structure. Here are some examples:

- Materials developed by the developer prior to or independent of the project. The materials are licensed to the customer, usually without restriction.
- Materials newly developed by the developer for the project. The intellectual property rights in these materials are usually assigned to the customer.
    - These may include modifications to third-party copyleft materials. The intellectual property rights in these modifications are usually assigned to the customer but may be effectively subject to third-party open source license conditions.
- Third-party open source materials. These are provided to the customer by the developer but licensed by the open source licensors.

Also, some developers today are hired specifically to create and release open source materials. In that case, the customer—rather than seeking an assignment of rights in the software—may want the developer to release the resulting development to the public under an open source license. The cautious customer will also receive a separate license to the material to ensure that it cannot be subject to an intellectual property claim for violating the open source license—which would be ironic after it paid for the development. Also, the customer may have an interest in ensuring that the developer seeks no patents on the software being developed. Accordingly, the customer may seek a promise to this effect or an assignment of the patent rights.

Because almost all development projects today are somehow related to open source, companies funding development are well served by paying close attention to the third-party open source that may be

used in development projects. Many companies unwittingly violate open source licenses when they engage developers who do not use proper open source compliance processes. A company should apply its open source compliance policies to outside developers just as it does to in-house developers.

It is a good practice to combine open source review and technical review in developer projects. Accordingly, many companies now include open source reviews in their acceptance processes.

Chapter 18

Government Regulation

Software is a largely unregulated market, so most of the rules of software licensing are a matter of private ordering, both in open source and proprietary models. However, in some tangential areas, government regulation and software licensing can come into conflict. This chapter discusses how to resolve—or at least balance—the rules of open source licensing and government regulation.

Government Procurement

The open source software movement embraces freedom and transparency. Open source software licenses use the power of copyright law to ensure that all users have the freedom to study, modify, and redistribute software. As the FSF says, "If you use a program to carry out activities in your life, your freedom depends on your having control over the program."[1] Laying open source code for all to see makes back doors, spyware, or other hidden control mechanisms easy to find and eradicate.

Freedom and transparency also underlie modern theories of government. As Justice Brandeis said, sunshine is the best disinfectant. Today, governments rely on technology to perform many basic functions. When software executes the functions of government, secrecy of source code is a political problem. Today, nearly every interaction between a citizen and the state involves a computer—whether that means citizens dealing with government agencies via

[1] www.gnu.org/philosophy/free-software-even-more-important.html.

websites, governments accessing one's electronic information by fiat, or governments using software and technology to conduct elections. Accordingly, many in and outside the open source movement believe that governments should employ open source software as much as possible to preserve freedom and transparency.

A government that buys its tools of operation with taxpayer funds also owes a duty of transparency to its citizens. In the United States, acquisition by governments or their agencies is governed by a web of procurement regulations. Federal procurement is governed by the Federal Acquisition Regulations (FAR), which govern acquisitions by all executive agencies.[2] Defense-related technology procurement is governed by FAR and its defense supplement, the Defense Federal Acquisition Regulation Supplement (DFAR). Acquisitions by states are governed by state procurement regulations, which are not necessarily consistent with the FAR or each other.

Although open source licenses certainly grant broad rights that can be useful to government, they do not always mesh with government procurement processes or requirements. Also, the open source development model may be inconsistent with the requirements of government procurement. For example, many jurisdictions have laws or procurement regulations requiring that goods purchased by the government be wholly or substantially manufactured within the government's territory of jurisdiction. Open source projects often solicit contributions from the general public, and contributors may be based anywhere. Also, the tendency of government to prefer cost-plus development may not mesh well with a project maintained by a group of volunteers.

In some areas, open source licenses either lack appropriate terms or conflict with government technology procurement requirements:

- **Governing law**. Some government procurement regulations require the application of local state law or federal law to the

[2] http://acquisition.gov/?q=browsefar.

terms and conditions of any contract. Most open source licenses purposely lack law or venue selection provisions.

- **Venue**. Many state and federal procurement regulations require that disputes be resolved in particular venues. Most open source licenses have no venue selection terms.
- **March-in rights**. Government funding of technology development under the FARs can entitle the government to compel a supplier to grant broad licenses to the technology.[3] Most open source licenses do not contemplate march-in rights, and such rights may have fewer restrictions than the applicable license.
- **Sovereign immunity**. A state government cannot be the target of a civil suit unless it has expressly waived its immunity. Contracting with state governments often involves a waiver of sovereign immunity to make agreements enforceable. In other words, by default law, no copyleft condition could be enforced against a state government unless the state government consents. Keep in mind that copyright law is federal law, and federal courts would not otherwise have jurisdiction over state governments.

Some of these issues could be addressed by supplementing existing open source licenses with side agreements for use when the licensee is a government agency. But this is not possible when the alternative term is a change to the scope or free exercise of the license, as opposed to ancillary agreements. For example, one might think that a government agency wishing to use GPL software could take the software under GPL and enter into an agreement with the licensor stating that the GPL will be interpreted under the law of the government's jurisdiction. GPL prohibits alterations of the license and also prohibits further restrictions on the license. Section 10 of GPL version 3 states, "You may not impose any further restrictions on the exercise of the rights granted

[3] 35 USC.§ 203: March-in rights.

or affirmed under this License."[4] Section 7 informs the licensee that if the software "contains a notice stating that it is governed by this License along with a term that is a further restriction," that notice may be ignored.[5]

There is much debate in the open source community as to what constitutes a *further restriction* prohibited by licenses like GPL, and FSF tends to take an expansive view of what this might include. For example, controversies persist over whether the Apple App Store terms (such as terms limiting use of software to a particular device) constitute an additional restriction and whether the patent defensive termination provision of Apache 2 makes it incompatible with GPL 2. Licensors and suppliers to government therefore take the risk of violating licenses like GPL if they agree to restrictions.

Moreover, even if the additional government-specific terms would not be considered additional restrictions or modifications in violation of a license, many government agencies, in practice, will not vary their procurement terms to accommodate open source licensing models.

Accordingly, as things currently stand, there is a fundamental inconsistency between copyleft licenses and government procurement, and it is unclear which side will be the first to blink. Governments continue to express a preference for open source software, but their policy objectives do not always mesh with their procurement rules.

Exports

In the United States, exports of software are covered by two sets of regulations: US Department of Commerce (part of the Bureau of Industry and Security or BIS) regulations and Department of Defense regulations. Software that includes encryption is subject to specific regulations.[6] However, all software exports are covered by the general

[4] www.gnu.org/copyleft/gpl.html.

[5] www.gnu.org/copyleft/gpl.html.

[6] Encryption products are regulated by the US Department of Commerce's Bureau of Industry and Security (BIS) through the Export Administration Regulations (EAR), 15 C.F.R. §§ 730–774. Exports of encryption products specifically designed for military applications are regulated

export regulations that restrict US persons from providing the products to others located in, or nationals of, countries on the list of embargoed destinations, or to specific entities on the Denied Persons List or Entity List.[7] Most open source software enjoys an exception to these regulations, for software for which source code is publicly available. Taking advantage of this exception requires a notification to the BIS of the source code's Internet location.[8]

This exception drew more attention than ever after Huawei, a top worldwide seller of mobile devices, based in China, was placed on the Entity List in 2019 as part of the trade war between the US and China, stemming from concerns about privacy and security. Huawei is an active participant in many open source projects, and those projects had to take appropriate steps to limit their disclosure of technology to Huawei to avail themselves of the relevant exceptions.

by the Department of State's Directorate of Defense Trade Controls (DDTC) through the International Traffic in Arms Regulations (ITAR), 22 C.F.R. §§ 120–130.

[7] https://www.bis.doc.gov/index.php/policy-guidance/lists-of-parties-of-concern/entity-list.

[8] 15 C.F.R § 734.7.and § 742.15(b). For more information, see https://www.eff.org/deeplinks/2019/08/us-export-controls-and-published-encryption-source-code-explained.

Chapter 19

Enforcement and Obstacles to Enforcement[1]

The twenty-first century has seen the first serious enforcement efforts by licensors of open source software, so we are truly at the dawning of the age of enforcement. But open source claims are not like other claims. Understanding the distinctions between open source software claims and other intellectual property claims is key to reacting to open source claims gracefully, effectively, and with a minimum of embarrassment and cost.

A survey of where we stand today demonstrates how the enforcement area has developed. We will soon be nearing the point where including catalogs of open source claims in books like this one will no longer be sensible or useful, but for now, seeing where we have been neatly explains where we are.

The early years of the twenty-first century saw the first written opinion on open source licensing law handed down in the United States. That decision, *Jacobsen v. Katzer*, was generally seen as a victory for open source licensing. It underscored the fact that open source licenses are not prima facie unenforceable, nor are they vulnerable to contract formation claims. Leading up to *Jacobsen* were a variety of claims—mostly settled long before they threatened to create case law—that set the groundwork for enforcement. Below is a list of most of the cases relating to open source licensing.

[1] An earlier version of this chapter appeared as "Open Source and the Age of Enforcement," *Hastings Science & Technology Law Journal*, September 14, 2012.

I. The Early Years: Pre-Jacobsen

A. Progress v. NuSphere (2001–2002)

NuSphere and MySQL had a business relationship in which NuSphere marketed several products that included both MySQL software and other proprietary software. Proprietary code (called Gemini) was statically linked to MySQL code in the NuSphere MySQL Advantage product. Although MySQL alleged a breach of its GPL license, the case was decided on trademark grounds, with the court sidestepping the GPL issues.[2]

B. Drew Technologies Inc. v. Society of Auto Engineers (2003–2005)

This was one of the earliest open source cases, filed in November 2003 and settled in early 2005. *Drew Tech* released certain software under the GNU General Public License (GPL), and that software was posted by an employee of Drew Tech on a message board run by the Society of Auto Engineers. Drew Tech sued to compel removal of the posting, and the case was settled and the posting removed. No reported opinion resulted, but the result suggested the GPL was enforceable.[3]

Although this case received little press and fanfare, it involved a familiar fact pattern, but in mirror image. Drew's employee posted the software without its GPL notices. The more common situation is the opposite. Today's technology companies often employ software engineers who are very enthusiastic about open source software. Those employees may, upon leaving the company or even before, make unauthorized code releases under open source terms that conflict with the proprietary terms offered by their company. This situation

[2] *Progress Software Corp. v. MySQL* AB, 195 F. Supp. 2d 328 (D. Mass. 2002). MySQL was historically licensed under the GPL plus "FLOSS" exception, which would make linking proprietary code to it non-compliant.

[3] For more detail, here is the *Groklaw* posting about the case: www.groklaw.net/articlebasic.php?story=20050225223848129.

sometimes is a result of a misunderstanding regarding who owns the code,[4] but it occasionally results from the actual malfeasance of disgruntled employees. In any case, unauthorized code releases are often resolved via a takedown of the code—as quietly as possible.

C. Jin v. IChessU (2006–2008)

Jin v. IChessU was a case that briefly whetted the appetite of open source lawyers because it squarely concerned the scope and interpretation of GPL—one of the big unresolved issues of open source law. In this case, International Chess University allegedly distributed the plaintiff's GPL software, a chess client called Jin, and added an audiovisual library to it, taking the position that the GPL did not require publication of the source code for the library. The author of Jin claimed it did. The case, which was brought in the courts of Israel, was settled in 2008.[5]

D. Planetary Motion, Inc. v. Techplosion, Inc. (2001)

This trademark case pointed to the use of GPL licensing notices as evidence of the trademark owner's intent to control the use of its mark.[6] The court said, "Because a GNU General Public License requires licensees who wish to copy, distribute, or modify the software to include a copyright notice, the license itself is evidence of [the licensor's] efforts to control the use of the 'CoolMail' mark in connection with the Software." Since 2001 when this case was decided, understanding of the relationship between open source licensing and trademark husbandry has developed significantly. Companies like Red

[4] Employees often do not fully understand the work-for-hire doctrine under copyright law, and they believe they own rights to open source software created during their employment "on their own time." The misunderstanding usually centers on the complexity of what constitutes the employee's own time and the breadth of assignments of rights in Employee Invention Assignment Agreements.

[5] For details, see www.jinchess.com/ichessu/.

[6] 261 F.3d 1188,; 2001 US App. LEXIS 18481; 59 USP.Q.2D (BNA) 1894 (11th Cir. 2001).

Hat have pioneered the husbandry of trademark rights parallel to open source licenses, rather than through them. Most commentators today would therefore consider GPL notices not to function as trademark notices. However, the court in this case was looking at the totality of the circumstances to assess whether the plaintiff had intended to use the mark as a trademark, and other facts and circumstances also contributed to the court's decision.

E. Computer Associates International v. Quest Software, Inc. (2004)

This opinion declares that "Bison is open source code, meaning that it is distributed by the FSF at no cost," that programs under the GPL are "freely released into the public domain," and that "[t]he GPL would prevent plaintiff from attempting to claim copyright in that modified version of Bison."[7] Again, understanding of open source licensing has developed significantly since this 2004 case, and so the court's comments should be viewed with skepticism; they represent an incorrect or, at best, an oversimplified view of open source licensing.[8]

F. gpl-violations.org (2004–2015)

This set of cases is usually considered the first significant enforcement effort for GPL. They were brought by gpl-violations.org, an organization spearheaded by Harald Welte, a technologist and open source advocate in Germany. That these cases preceded actions in the United States is not surprising; the German legal system differs substantially from the US legal system. For instance, German courts allow ex parte actions for injunction—a suit for an injunction brought by the plaintiff where the defendant does not participate and the plaintiff pursues the injunction directly with a court.[9] Some, but not all,

[7] 333 F. Supp. 2d. 688, 697–98; 2004 US Dist. Lexis 11832 (N.D. Ill. 2004).

[8] My thanks to Terry Ilardi for pointing me to the *CAI* and *Planetary Motion* cases.

[9] Germany is a jurisdiction particularly friendly to the granting of intellectual property injunctions. See http://www.lexology.com/library/detail.aspx?g=d25319ed-026f-4a88-a7d5-1ab4f24db60c.

of the initial string of cases brought by this organization related to Mr. Welte's own software. The most widely publicized of these cases was against Fortinet UK. A Munich district court granted a preliminary injunction against Fortinet prohibiting distribution of their products absent compliance with the GPL. Welte also claimed that Fortinet was obfuscating the existence of GPL code in its product, a fact Fortinet disputed. Fortinet eventually agreed to make certain source code in its products available under GPL.

Since that time, other authors empowered gpl-violations.org to bring claims on their behalf in a similar fashion, and gpl-violations.org grew in scope to an ad hoc enforcement organization for GPL and other free software, working with FSF Europe.[10]

II. Formal Enforcement in the United States Bears Fruit

A. The BusyBox Cases (2007–Present)

Starting in 2007, the Software Freedom Law Center (SFLC) filed a series of lawsuits on behalf of Erik Andersen and Rob Landley, two of the authors of the BusyBox software. BusyBox emulates many standard UNIX tools in a small, efficient executable that's useful for embedded devices. The first suit was filed against Monsoon Multimedia, Inc. That case was settled with the release of source code and an undisclosed settlement payment. SFLC next brought suit against Xterasys and High-Gain Antenna. Both soon settled. Further suits were brought from 2007 to 2009 against Verizon Communications, Bell Microproducts, and Super Micro Computer. All settled quickly.

On December 14, 2009, SFLC filed another lawsuit naming fourteen defendants, including Best Buy, JVC, and Samsung. Some of these are still pending.

[10] The website shut down in early 2015; see http://en.wikipedia.org/wiki/Gpl-violations.org.

One case, against Westinghouse Digital Electronics, resulted in a damages award, but this was because the defendant was in liquidation and defaulted on the litigation by failing to answer discovery requests. The judge awarded statutory damages, attorneys' fees, and injunctive relief.[11] Although the order characterizes the damages as "treble," that was only true in the sense that the court awarded three times the upper limit of statutory damages in the absence of willfulness. The better characterization is enhanced damages—under 17 USC § 504, a court can in its discretion award up to $150,000 per work in statutory damages if the infringement is willful. The court did not make a finding of actual damages because the defendant failed to respond to discovery and thus never presented evidence to support calculation of actual damages. Also, as described in footnote 39 of the order, absent the default, statutory damages might have been unavailable because the plaintiff had failed to register the copyright in a timely fashion. Given that most software authors do not register their copyrights, this could often be a significant limitation on claims enforcing open source licenses. This case's result is interesting mostly because it confirms that injunction and statutory damages are available for open source claims, but the facts were unusual and thus probably difficult to extrapolate to non-default judgments.

In a further development in the same case, a successor entity of Westinghouse Digital Electronics met the same fate. On August 8, 2011, the District Court for the Southern District of New York held Westinghouse Digital, LLC ("WD") in contempt for failing to defend its BusyBox case. Westinghouse Digital Electronics, LLC ("WDE"), with the default judgment standing against it, liquidated its assets after severe business distress and told the court it would not defend the litigation. WD purchased the assets of WDE. The court assessed whether there was "a substantial continuity of identity" between WDE and WD, and it found that there was. The court invited evidence on damages and attorneys' fees and ordered forfeiture of all infringing articles.

[11] Attorneys' fees were awarded because of the default.

It is important to note that the BusyBox cases involved only some of the developers of BusyBox and notably did not include Bruce Perens, the original author, and Dave Cinege, the maintainer who had made significant contributions to BusyBox. Perens subsequently released a statement criticizing the current BusyBox developers, saying, "The version 0.60.3 of BusyBox upon which Mr. Andersen claims copyright registration in the lawsuits is to a great extent my own work and that of other developers. I am not party to the registration. It is not at all clear that Mr. Andersen holds a majority interest in that work."[12]

Joint ownership is a lurking issue in open source enforcement. While in the case of BusyBox the primary authors were more or less sequential (the laboring oar having been passed from Perens to Cinege, and so forth), open source projects are often collaborative efforts. The messaging and rhetoric regarding open source development often emphasize open source's collaborative nature, and this is prime fodder for an argument that open source projects are actually joint works. Under US law, joint authors all have an undivided interest in the work and can freely license it.[13] Therefore, defendants facing claims of a single author among many may be in a position to seek a license from a joint author and claim a license defense. Alternatively, defendants may be able to raise procedural arguments that all authors must be joined to the suit for the claim to go forward. US courts sometimes require this and sometimes do not. When they do, the reason given is that if any author could have granted a license, then the claim cannot be resolved until they are all involved. Even if a defendant is not ultimately successful in claiming a license from a non-plaintiff author, it may be able to delay and complicate the suit enough to make enforcement difficult.

The BusyBox cases in many respects comprise the quintessential action of the age of enforcement. They are brought by SFLC, albeit on behalf of private parties. Their settlement tends to be quick, reflecting SFLC's strategy of establishing a track record of enforcement by

[12] http://lists.gnu.org/archive/html/fsf-community-team/2009-12/msg00127.html.

[13] Subject to certain requirements to account, for instance. See *Oddo v. Ries*, 743 F.2d 630 (9th Cir. 1984).

selecting clear violations to pursue. The press releases announcing the terms of these settlements read like a template and almost always describe the following in some detail: appointment of an open source compliance officer, covenants to bring distribution into compliance (via communication of notices and release of source code), and the payment of damages—though probably less than one would expect from a proprietary software lawsuit. This form of settlement reflects the quintessential concerns of a zealous plaintiff—with a focus on compliance and transparency, rather than substantial damages.

B. Jacobsen v. Katzer (2006–2010)

In August 2008, the US Court of Appeals for the Federal Circuit issued the most significant decision on open source licensing in the United States to date—and followed the guidance of the open source community.

The case arose from a complicated set of facts. Both parties developed and distributed software for controlling model railroads. Jacobsen made his "Java Model Railroad Interface" (JMRI) software available under an open source license free of charge, and Katzer (via his company Kamind Associates) sold commercial products under proprietary licenses. In particular, Jacobsen made available through the JMRI project software called DecoderPro, which is used by model railroad enthusiasts to program decoder chips in model trains to control lights, sounds, and speed.

Jacobsen received a letter inviting him to license patents owned by Kamind, suggesting the patents were infringed by the JMRI software. Jacobsen filed a declaratory judgment action asking the court to rule that the patent was invalid due to prior art (or failure to disclose prior art, including that of Jacobsen himself) or not infringed. Katzer's original claim received a great deal of press in the open source world, feeding fears that the existence of software patents spelled death for open source projects. As the patent case progressed, Jacobsen discovered that Katzer had copied some of Jacobsen's open source software and used it in Katzer's proprietary product. Jacobsen's software was licensed under the Artistic License, which is a relatively rarely used license that

was originally written to provide rights to the PERL programming language interpreter. The requirements of the Artistic License are modest; it is generally considered a permissive open source license, similar to the Apache, BSD, or MIT licenses. The Artistic License requires, as a condition to exercise of the license, certain copyright and license notices and identification of changes made to the original author's source code. Because these notices were not preserved by Katzer, Jacobsen made a counterclaim, alleging violation of the license.

In 2007, the District Court for the Northern District of California issued a preliminary ruling in the case, stating that the license violation constituted a breach of contract. However, the court's ruling did not support a claim for copyright infringement. The court reasoned that Katzer was exercising rights in the copyrightable work but was licensed to so do and thus was not liable for infringement. Copyright infringement claims can result from breach of the scope of a license, but the court distinguished violations of license scope and license conditions. This distinction is core to open source law because open source licenses grant all the rights of copyright—and so it is generally not possible to violate the scope of an open source license, only its conditions.

Proponents of free software such as the Free Software Foundation have long taken the position that their licenses are not contracts. Originally, this position may have been a strategic attempt to avoid the contract formation issues that plagued the law of online distribution of the software in the early 1990s, but the position persists notwithstanding the intervening publication of a line of cases supporting enforceability of contracts in the download context.

The issue in this case is sometimes referred to in the open source legal community as the *license or contract* issue. In other words, are open source licenses contracts, or are they merely conditional licenses? (See Chapter 5: Conditional Licensing, for an extended discussion of this issue.) The status of an open source license as a license or contract dictates the types of remedies available if a licensee violates the conditions of the license. If the conditions of an open source license are mere contractual covenants, as the district court order stated, then

injunctive relief is generally not available. If, however, violating the conditions places the activities of the licensee outside the scope of the license, as the appellate court stated, then the unlicensed activity is copyright infringement and far more likely to garner injunctive relief. Open source advocates are also concerned that the money damages for violation of open source licenses under contract law may be more limited than those available under copyright law, which can include statutory damages as well as actual damages.

The Free Software Foundation has long taken the position that open source licenses are licenses rather than contracts—however, this can be misleading because the two are not mutually exclusive. Most licensing contracts are both conditional licenses and contracts. The *Jacobsen* case holds that violating the conditions of an open source license can constitute copyright infringement but not that open source licenses are not contracts.

The flip side of this question—whether the conditions of a license could be considered covenants whose performance might be ordered by a court—was not at issue. But this is the "bet the company" issue for most corporate free software users. If providing source code were a contractual covenant, then failure to do so would be a breach of contract. However, even if it were a breach, specific performance of covenants is not generally available under contract law. Most companies' biggest fear when dealing with GPL is that they will be compelled to lay open proprietary software. In fact, the likely worst-case scenario is that they will be given a Hobson's choice: to lay it open and comply with GPL, or to replace the GPL code.

On the question of whether the claim could sound in copyright—that is, whether it could be cast in terms of copyright—the appellate court reversed and remanded the case back to the district court to determine whether Jacobsen had established his claim to an injunction. The Federal Circuit opinion was hailed as a victory for open source licensing.

Ironies abounded in this case. This decision, possibly the most significant in open source software licensing to date, relates to the relatively obscure Artistic License—though the court's mention of the

GPL in a footnote suggests it intended the same result to hold for other open source licenses. Moreover, the decision was issued by the US Court of Appeals for the Federal Circuit, which is the circuit that primarily adjudicates the enforcement of patents—the villains of the open source world. Finally, the case was only brought because Katzer chose to pursue a patent claim against a party whose code he had apparently copied—a strategy probably best avoided.

On December 10, 2009, on remand, the lower court issued an order granting and denying portions of motions for summary judgment by the parties. The court said, "Although it is undisputed that Plaintiff distributed the copied work on the Internet at no cost, there is also evidence in the record attributing a monetary value for the actual work performed by the contributors to the JMRI product." As a result, the court said that the record "may establish a monetary damages figure." This dispels the notion that no actual damages would be available for open source authors bringing claims of infringement because of the royalty-free nature of open source licensing.

Jacobsen v. Katzer was finally settled in 2010, obviating a second appeal. The settlement included an injunction by the district court against further infringing activities, and it was not sealed. Most settlements are confidential, and an actual injunction by the court in a settlement is a bit unusual. But the injunction would have carried more precedential weight if it had been won in court rather than agreed to in settlement. (The District Court had previously declined to issue an injunction.)

It was significant that this case involved the original Artistic License. If a permissive license is enforceable, that lays substantial groundwork for enforceability of copyleft licenses. In *Jacobsen*, the question of actual harm to the copyright owner was squarely at issue. For Jacobsen to prevail, the court had to decide that there was sufficient harm to give rise to a remedy. With a permissive license, the harm is, essentially, failure to deliver notices. For a copyleft license, the conditions are much more significant, and violating them would give rise to greater harm and, in turn, greater opportunities for relief.

C. FSF v. Linksys (2008–2009)

This suit was notable in that it was the first lawsuit filed for FSF as a plaintiff. Compliance issues regarding GPL software in Linksys consumer wireless routers had existed since Cisco acquired Linksys in 2003. (For details about the informal dispute that took place in the early 2000s, see my 2005 article "The Legend of Linksys" in *Linux Insider*.[14]) This dispute resulted in a fairly quick release of source code, but FSF continued to find compliance issues with the routers and finally filed suit in 2008. The accused software included various GPL and LGPL components.[15] That suit was settled in 2009 with no substantial docket activity via a settlement that, according to the press release describing it, was very similar to those in the BusyBox cases.[16]

III. Post-Jacobsen and Strategic Plaintiffs

A. Artifex Cases (2008–Present)

Artifex is the purveyor of the Ghostscript line of software, which is primarily used as embedded software in printers. In 2008 and 2009, Artifex filed some of the first lawsuits brought by a private company to enforce GPL. The first plaintiff was Premier Election Solutions, at the time a subsidiary of Diebold. That case was settled quickly and confidentially. In 2009, in an action related to its MuPDF software (a high-performance PDF-rendering engine), Artifex brought claims against Palm, Inc. and various other defendants. The cases have settled.

[14] www.linuxinsider.com/story/43996.html?wlc=1300413418.

[15] See the complaint and FSF's press release here: www.fsf.org/blogs/licensing/2008-12-cisco-complaint.

[16] www.fsf.org/news/2009-05-cisco-settlement.html.

B. Twin Peaks Software, Inc. v. Red Hat, Inc. et al. (July 2013)

Twin Peaks Software (TPS), which made proprietary network backup software, sued Red Hat and Red Hat's recently acquired subsidiary, Gluster. TPS claimed that the GlusterFS software—a network file system that aggregated multiple storage shares into a single volume—violated TPS's patent covering TPS's Mirror File System (MFS). Red Hat initially responded to the patent infringement suit by denying the infringement and asserting that the patent was invalid, but it later brought a counterclaim alleging the TPS products incorporated open source software from Red Hat's product but failed to comply with GPL. Red Hat sought an injunction against the TPS products. The case soon ended in a settlement, suggesting that TPS thought better of pursuing its patent claims in light of the facts.

C. US Customs Case (Unfiled)—Fusion Garage

In September 2010, a Linux kernel developer, Matthew Garrett, posted a blog entry[17] threatening to file a US Customs case based on GPL violations. His blog describes failed attempts to get the source code for JooJoo Android tablets. According to this blog, the maker of the tablet, Fusion Garage, has not responded to requests for source code as required by GPL.

This method of enforcement is sometimes called a *337 action* because it is authorized by Section 337 of the Tariff Act of 1930 (19 USC. § 1337). If the US ITC (US International Trade Commission) finds Section 337 has been violated, it may issue an order directing that infringing goods be excluded from import into the United States. The order is executed by Customs and Border Protection (CBP), which may seize the goods at the border. CBP's border enforcement of copyrights

[17] http://mjg59.livejournal.com/126865.html.

is essentially limited to copyrights that have been registered with the US Library of Congress and also recorded with CBP.

A circular published by the US government says this:

> Members of the public may inform CBP of potential intellectual property rights violations via CBP's on-line trade violation reporting mechanism called e-Allegations. The public may access e-Allegations and additional relevant information at www.cbp.gov/trade/trade-community/e-allegations/e-allegations-faqs/. CBP also maintains an on-line recordation system, Intellectual Property Rights e-Recordation, which allows rights owners to electronically record their trademarks and copyrights with CBP and facilitates IPR seizures by making IPR recordation information readily available to CBP personnel.

This means that while only the owner of the copyright can bring a claim, anyone can report an infringing import. These 337 actions have long been used as a tactic to enforce intellectual property rights. They are generally faster and cheaper than federal litigation, and for this reason they are particularly popular with patent plaintiffs. However, the remedies available are different from those in federal court—for example, a quick emergency exclusion order is more likely, but damages are not available. In addition, in a patent action, the complainant must show that the patent is being used in an "existing domestic industry" (19 USC. 1337(a)(1)(B), (a)(2–3)). This may exclude actions by nonpracticing entities (NPEs) or patent trolls.

Since Garrett's blog post, nothing appears to have been filed, but the case is still worth mentioning because it shows another approach to litigation in the open source sphere: using a tactic already popular for patent and other intellectual property claims.

D. Microsoft's Linux Patent Enforcement Program (2009–Present)

In 2009, Microsoft filed a patent lawsuit against TomTom, makers of consumer GPS devices, accusing code in the Linux kernel of infringing certain patents of Microsoft, including three file management patents. The suit was settled very soon thereafter, with payment of a license fee to Microsoft and a five-year patent cross license

between the parties. A joint press release called the settlement "fully compliant with TomTom's obligations under the General Public License Version 2." TomTom also agreed to remove from its products the functionality related to two file management systems patents over a two-year period.[18]

The patent claims asserted in the TomTom case are also the basis of a string of enforcement activities by Microsoft, which has clearly identified Linux as a competitive threat. Virtually all the enforcement has been in the consumer electronics space: I-O DATA, Amazon.com, Novell, Brother International Corp., Fuji Xerox Co. Ltd., Kyocera Document Solutions Corp., LG Electronics, and Samsung Electronics Co. Ltd.

Although these claims are not strictly related to open source licensing—the outbound license of the accused product being irrelevant to the central patent infringement questions of infringement and validity—they bear upon open source licensing in two ways. First, Microsoft's form of settlement was tailored to conform to the defendant's licensing of the accused software under GPL 2 but not GPL 3, which contains provisions that would contradict the terms of most such settlement agreements.[19] Second, Microsoft is probably one of the very few technology product companies that does not distribute Linux—if it did, it might have significant challenges enforcing patents that read on GPL code.[20]

Microsoft's enforcement actions are ongoing.

[18] http://news.microsoft.com/2009/03/30/microsoft-and-tomtom-settle-patent-infringement-cases/.

[19] See the terms of paragraphs 5, 6 and 7 of GPL 3 Section 11—the so-called "anti-Microsoft" and "anti-Novell" provisions.

[20] The topic of what patent rights are licensed under GPL 2 is a subject of controversy—or better said, mystery—but is beyond the scope of this paper. For more information, see Adam Pugh and Laura Majerus's 2006 paper: http://www.fenwick.com/FenwickDocuments/Patent_Rights.pdf.

E. Oracle America v. Google (2010–Present)

In 2010, Oracle, after having acquired Sun Microsystems (and renaming it Oracle America, Inc.), filed suit alleging that Google's Android mobile platform infringed certain patents of Oracle America, as well as copyrights in Oracle's Java platform. Oracle licenses Java under, among other terms, GPL plus a "Classpath exception."[21] In this suit, Oracle alleged that Google's reimplementation of portions of the Java API infringed copyrights associated with Java. The district court found for Google on all patent and copyright claims. The court ruled, "So long as the specific code used to implement a method is different, anyone is free under the Copyright Act to write his or her own code to carry out exactly the same function or specification of any methods used in the Java API. It does not matter that the declaration or method header lines are identical."[22] Therefore, Google's actions did not require a copyright license from Oracle; under section 102(b) of the Copyright Act, the API was a "system or method of operation" that did not enjoy copyright protection. The case therefore did not, ultimately, turn on whether the GPL had been violated. The case was appealed to the Federal Circuit, which reversed, and a petition for certiorari was denied by the US Supreme Court. The case was subsequently retried on the issue of fair use, and in May 2016, the jury unanimously sided with Google, concluding that Google's reimplementation of Java APIs was fair use. As this book goes to press, Oracle's appeal of the judgment is pending to the Supreme Court, after a reversal by the Federal Circuit on the fair use verdict.[23]

F. Penrose Trademark Dispute (2011)

A complaint was filed against Red Hat in the Northern District of California on May 6, 2011, alleging various claims, including a request to

[21] The Groklaw site (www.groklaw.net) contains extensive information about the case.

[22] *Oracle Am., Inc. v. Google Inc.*, 872 F. Supp. 2d 974 (E.D. Cal. 2012).

[23] For the Electronic Frontier Foundation's current summary, see www.eff.org/cases/oracle-v-google/.

cancel a trademark registration. According to the complaint, Alex Karasulu, founder of the Apache Directory Server Project, set up a domain at www.safehaus.org to provide an ecosystem for open source software components related to directory and security infrastructure. Karasulu was approached by Jim Yang about the development of open source virtual directory software, and Karasulu offered to develop the software as a project for Safehaus. Karasulu used the name "Penrose" for the virtual directory software project. The initial version of the Penrose software was released on May 23, 2005. On March 13, 2008, Yang filed an application to register the trademark Penrose for software through a company he owned called Identyx, Inc. Yang subsequently sold Identyx to Red Hat. The complaint alleges that in the trademark application, Yang misrepresented the facts underlying the first use of the mark and failed to acknowledge prior use and ownership by Safehaus, and it also alleges that Red Hat knew statements made in the trademark application by Identyx were false because of information received by Red Hat in the due diligence process for the acquisition. Based on the above allegations, the complaint alleged various claims and asks the court to declare the trademark registration invalid.

G. Lawsuit by gpl-violations.org Against FANTEC (2013)

Gpl-violations.org filed a successful claim in Germany against FANTEC, a European company selling devices that facilitate streaming of media content.

One of FANTEC's products, the FANTEC 3DFHDL Media Player, included firmware based on the Linux operating system. Linux includes many software components covered by the GNU GPL 2, which requires that any distributor of binaries make the source code available to binary recipients. To comply with the GPL, FANTEC made a version of the source code available for download that it had received from its Chinese contract manufacturer. Unfortunately, it was not the right source code for the binaries.

Harald Welte brought the action as one of the authors of iptables, a packet-filtering utility licensed under GPL. FANTEC had previously settled a GPL dispute with Welte in 2010 by agreeing to a cease-and-desist declaration that provided for contractual penalties if FANTEC committed any future GPL violation. At a 2012 Hacking for Compliance workshop hosted by the Free Software Foundation, compliance engineers discovered that the firmware object code shipping with the 3DFHDL included iptables and that the source code provided by FANTEC did not. This information was forwarded to Welte, who gave FANTEC notice of the violation and demanded that FANTEC pay the contractual penalty set forth in the cease-and-desist declaration. FANTEC responded that it had been assured by its contract manufacturer that the source code was complete, and it refused Welte's demand. Welte sued FANTEC in the Regional Court of Hamburg, seeking to enforce the cease-and-desist letter.

FANTEC claimed that analyzing the source code would have been a costly process whose results may not have been reliable, and it argued that because only the author could determine with certainty whether the source code was accurate and complete, FANTEC should be permitted to rely on the representations of its contract manufacturer, who provided the firmware. However, the court disagreed and held FANTEC responsible for the error.

This case is a cautionary tale about a very common problem. Like many consumer electronics companies, FANTEC was caught between its customers and its supply chain. In ruling for Welte, the court held that a distributor of software may not rely on assurances made by the supplier of the software that the software does not infringe the rights of any third party. According to the court, FANTEC was "culpably negligent" in distributing incomplete source code. The court held that FANTEC was responsible for determining, "by [its] own assessment or with the help of a qualified third party," that the software did not infringe any third-party rights, even if doing so would have caused FANTEC to incur additional costs.

An open source distributor cannot rely on the culpability of its suppliers to avoid liability. It is best to always make delivery of source

code, as well as a demonstration that it corresponds to the binaries, part of the acceptance testing for deliverables from one's supply chain.

H. Mein Büro (2013)

Adhoc Dataservice GmbH settled with Buhl Data Service GmbH, the developer of the WISO Mein Büro (My Office) enterprise application software. Buhl agreed to pay €15,000 for violating the LGPL terms under which Adhoc provides its FreeadhocUDF open source library (which includes various functions for UDF). The settlement disposed of a court case that resulted in a January 2011 opinion from a lower court in Bochum, Germany that the use of the FreeadhocUDF library in WISO Mein Büro 2009 violated the LGPL's licensing terms.[24]

I. Continuent, Inc. v. Tekelec, Inc. (2013)

In this recent case involving a GPL enforcement claim in the United States, a private company sued to enforce the GPL. The complaint, filed on July 2, 2013 in the US District Court for the Southern District of California, alleged that defendant Tekelec copied and distributed Continuent's software without permission and in violation of GPL 2, thus infringing on Continuent's copyright. The case settled and was dismissed.

J. Versata/Ameriprise/Ximpleware (2014)

A series of cases recently raised issues relating to distribution, the scope of GPL, and remedies for violation of GPL. For a good summary of these cases, see opensource.com/law/14/7/lawsuit-threatens-break-new-ground-gpl-and-software-licensing-issues. However, those cases have now settled and been dismissed.

[24] For the German decision, see www.telemedicus.info/urteile/Urheberrecht/Open-Source/1148-LG-Bochum-Az-I-8-O-29309-Ansprueche-bei-Verletzung-der-LGPL.html.

K. Hellwig v. VMware (2015 - 2017)

Christoph Hellwig, a significant contributor to the Linux kernel, brought suit in the German district court in Hamburg against VMware Inc. for failure to comply with GPL 2 in VMware's ESXi products. Legal commentators in the open source area have been watching this case closely because it raises issues of the scope of GPL.

The case involves the opposite of the quintessential GPL 2 border dispute questions. For many years, open source lawyers and engineers have been divided over whether it is compliant to add proprietary drivers to the GPL 2–licensed Linux kernel. But in this case, VMware's proprietary ESXi kernel, which enables virtualization, was capable of being built with Linux drivers.[25] The court stated that Hellwig had failed to show that he had sufficient authorship to sustain the suit. Specifically, the court found inadequate Hellwig's attempt to establish authorship of his contributions via information in a GIT repository—a problem that may spell trouble for future GPL enforcement cases by Linux kernel developers. As this book goes to press, Hellwig has announced that he has discontinued his case.[26]

The Hellwig case generated a fair amount of discord in the open source community over whether legal action to enforce GPL—much less in a court that was perceived by some as chosen by venue shopping—was good for free software development. But it seems inevitable that SFC will continue to pursue legal remedies when it feels doing so is necessary, and Germany remains a venue of choice for GPL enforcers.

[25] Software Freedom Conservancy (SFC) believes this was noncompliant; its technical explanation is here: http://sfconservancy.org/copyleft-compliance/vmware-lawsuit-faq.html. An unofficial translation of the dismissal order appears here: http://bombadil.infradead.org/~hch/vmware/Judgment_2016-07-08.pdf.

[26] https://sfconservancy.org/news/2019/apr/02/vmware-no-appeal/.

L. McHardy v. Various Parties

Starting in about 2015, a kernel developer named McHardy began bringing lawsuits against multiple parties in Germany for violation of GPL. Because Germany allows such suits to be brought in a non-public way, it is impossible to verify how many suits were brought or against which parties. It has been generally known in the industry for a while that this was occurring, but confidentiality obligations have prevented open discussion of the lawsuits.

McHardy is the former chair of the Netfilter core development team. Netfilter is a utility in the Linux kernel that performs various network functions, such as facilitating Network Address Translation (NAT)—the process of converting an Internet protocol address into another IP address. Controlling network traffic is important to maintain the security of a Linux system.

The silence was broken, in part, when the Netfilter project posted that it "regrets to have to suspend its core team member Patrick McHardy" due to allegations regarding "the style of his license enforcement activities on parts of the Netfilter software he wrote." The project did not disclose the allegations or their targets. The project pointed out that it "does not have first-hand evidence" but cited "various trusted sources."

SFC then made a stronger statement connecting the dots between promulgation of its community enforcement guidelines, the adoption of those guidelines by the Netfilter project, and the subsequent suspension of McHardy. SFC was clearly trying to distance itself from McHardy's tactics.

McHardy, like Hellwig, has experienced difficulty in sustaining his claims due to procedural issues, and it is generally thought that the quality of his advocacy is not high. That is, the parallels drawn between McHardy, who has been pejoratively termed a "copyright troll," and patent trolls who bring suits using marginal counsel and employ a strategy of seeking quick settlements, may be apt ones. This trolling strategy may be facilitated by the kind of order that was issued in the case (unnamed) discussed immediately below, which imposes heavy

fines for future violations—a mechanism that could be very financially beneficial to a plaintiff.

Organizations like SFC have a deep interest in preserving the credibility and mission of enforcement actions. However, licenses like GPL are designed to give authors the power to bring copyright claims for noncompliance, and copyright law is a sharp sword. So those who seek to enforce for pecuniary gain will always have the legal right to do so, and community norms may not convince them to forego that right.

McHardy's trolling actions are possible in part due to plaintiff-favorable elements and practices in the German legal system. For example, McHardy has issued a communication called an Abmahnung ("warning"): The "warning" is a request from the claimant to the defendant to stop doing something. In the copyright context, it is a letter from the copyright owner requesting that an alleged infringer stop infringing. These letters are issued by lawyers, not courts, and are often the first step in a copyright enforcement action in Germany. In the U.S., the closest analog would be a cease and desist letter.

Warnings often also include an Unterlassungserklärung ("cease and desist declaration" or "declaration of injunction"). This "declaration" is like a contract — signing it subjects the defendant to legal obligations that might not otherwise exist. In particular, the declaration may contain obligations that are not required by the GPL itself. In Germany, it is common for such a document to contain penalties for noncompliance. In the US, the analog would be a settlement agreement, but settlement agreements rarely specify the penalties for breach — and in fact, in the US, "penalties" may not be enforceable in contracts. A cease and desist declaration may also contain a non-disclosure requirement, and that can curtail the ability of a defendant to seek support from other defendants or to alert the community about the claimant's assertions.

The tactic of using cease and desist declarations with substantial penalties can be a powerful weapon for copyright trolling in the German legal system.[27]

M. Preliminary Injunction for Violation of GPL (July 2015)

A German court (the regional court in Halle) issued a decision interpreting the GPL, along with a cease-and-desist order against further non-compliant use of the licensed software.

The software was licensed under GPL 2 or any later version. The defendant, a German university, made the software available for download by its staff and students, without providing GPL notices or source code. The plaintiffs complained to the university, and the university removed the licensed software from its servers. The plaintiff then approached the university, asking it to sign a "cease and desist declaration with a penalty clause"; the defendant refused to sign, claiming that doing so was unnecessary because the software had been taken down.

Although the court did not specifically say which version of GPL (2 or 3) it was interpreting, the dispute centered on the termination provisions of the license. Section 8 of GPL 3 retained the automatic termination in GPL 2, but provided for reinstatement of rights if the licensee cured its breach within 30 days. In this case, the defendant cured its breach within the necessary period but refused to sign the cease-and-desist declaration. Nevertheless, the court ruled the reinstatement provision did not eliminate the plaintiff's right to a preliminary injunction to prevent further infringements. The court order also provided for a fine of between 5 and 250,000 euros (and if not paid, imprisonment of up to six months) for continuing violations, as well as payment of legal costs.

[27] For more detailed information, see https://blog.fossa.io/patrick-mchardy-and-copyright-profiteering-44f7c28c0693.

This case demonstrates the plaintiff-friendly nature of the German venue, as well as significant differences between US and German law. In the United States, an injunction would not be available to prevent recurrence of a violation that has been terminated, and failure to pay fines would not ordinarily result in jail time.

IV. Other Open Source–Related Cases and Disputes

Several notable cases did not strictly involve open source licensing issues but had an impact on perceptions of intellectual property risk in open source.

A. SCO v. IBM, Novell, Red Hat, AutoZone and Daimler-Benz (2003–Present)

While this set of related cases was by far the most publicized in the open source world, the SCO cases were primarily breach-of-contract cases and did not seek to enforce any open source license. In fact, IBM raised SCO's breach of the GPL as a counterclaim, but this series of cases was resolved on different facts. A jury verdict in April 2010 determined that SCO did not own the copyright in UNIX, all but killing SCO's claims. The true end will be dismissal of SCO's case against IBM, which was stayed pending the disposition of SCO's bankruptcy, which will in turn depend heavily on the result in the Novell suit. This case, while not exactly over, is in the process of dying a long, slow death.

B. MontaVista Software v. Lineo (2000)

MontaVista sued its competitor, Lineo, claiming that Lineo was distributing software written by MontaVista with the copyright

notices removed.[28] The case was settled. The issue in this case—the stripping of notice—was not particular to open source.

C. Monotype v. Red Hat (2003)

Monotype sued Red Hat claiming copyright and trademark infringements. The suit was settled, and in December 2003, the parties entered into a license agreement that provided Red Hat the right to distribute certain Monotype commercial fonts over a five-year period. The license cost Red Hat $500,000. This was not a suit to enforce any open source license; it was an infringement suit regarding proprietary software.[29]

D. MDY v. Blizzard (2010)

This case did not involve open source software, but it is notable for its possible tension with *Jacobsen v. Katzer*. This case involved the creation by MDY of Glider, a "leveling bot" for use in Blizzard's *World of Warcraft*. MDY sold copies of this bot to users, causing concerns for Blizzard. Blizzard added a provision to its terms of use for *WoW*, prohibiting the use of bots. This case was an action for declaratory judgment by MDY, claiming that its exploitation of the bot did not infringe Blizzard's copyright in the *WoW* software. The lower court awarded damages for contributory copyright infringement, based on MDY's customers' use of the bot. The Ninth Circuit reversed, stating that the violation of the bot prohibition did not have a sufficient nexus to copyright to support a claim for infringement. In so ruling, it interpreted the prohibition as a covenant rather than a condition, citing relevant state law. (In contrast, the Federal Circuit in *Jacobsen* did not defer to state law, viewing the case as an intellectual property claim.)

[28] Steven Shankland, "Linux Companies Settle Copyright Suit," C/Net News.com, October 13, 2003, http://news.cnet.com/Linux-companies-settle-copyright-suit/2100-7344_3-5090704.html.

[29] See Red Hat's Annual Report on Form 10-K dated February 29, 2004, www.secinfo.com/d14D5a.12Mg6.htm.

E. SAS Institute Inc. v. World Programming Ltd. (2010)

This case is not strictly about open source software, but it addresses the issue of whether APIs are copyrightable. The Royal Court of Justice issued an opinion on July 23, 2010.[30] SAS Institute Inc. is a developer of the SAS system, which allows users to write scripts to manipulate and analyze statistical information. World Programming Ltd. created a product called World Programming System, which was designed to emulate much of the functionality of the SAS system as closely as possible via the same programmatic interface. WPL had no access to the source code of the SAS system. SAS alleged that World Programming had infringed the copyright in the interface and documentation. The court referred questions of protectability of programmatic interfaces under copyright law to the European Court of Justice, noting that "national courts must interpret both European and domestic legislation as far as possible in the light of the wording and purpose of relevant international agreements to which the EU is a party, such as TRIPS and the WIPO Copyright Treaty."

The copyrightability of APIs is a potentially significant issue to open source licensing because the Free Software Foundation's position—that linking of new software to GPL software creates a derivative work of the original GPL software—depends on the ability of an author, under GPL, to protect the interface to software based on copyright law. Also, many open source software projects (such as Linux) are reimplementations of proprietary software that have been reverse engineered from their APIs.

F. FFmpeg Trademark Dispute

The FFmpeg project—a very popular open source implementation of MPEG codecs—posted a note about a legal threat that concerned the logo for the project.[31] FFmpeg says the threat comes from "a

[30] www.bailii.org/ew/cases/EWHC/Ch/2010/1829.html#para29.

[31] Originally at http://www.ffmpeg.org/%EF%BB%BF. The posting has since been taken down.

previous root admin of FFmpeg, who now is root admin of the Libav fork of FFmpeg." The threat claimed a copyright on the diagonal-line logo. FFmpeg replaced it with a new version that represented the same shape with different shading and contours. FFmpeg posted an email suggesting that the claimant did not come up with the original shape, only a particular rendering of it. This is a garden variety trademark dispute, unrelated to the open source nature of the project.

G. Novell/CPTN/Attachmate Antitrust Issues

In November 2010, Novell announced its acquisition by Attachmate for $2.2 billion. Novell retained its copyrights to UNIX following the merger.[32] Novell separately sold 882 patents to CPTN Holdings, a technology consortium led by Microsoft, for $450 million. The deal caused consternation over whether the sale of the patents was a threat to Linux.

However, Novell's patents were apparently already encumbered in favor of Linux at the time of the sale. Novell was a founding member of the Open Invention Network (OIN), which would have made its patents subject to OIN's patent policies. OIN is essentially a patent commons for Linux; members agree to enter into cross-licenses of their patents to cover Linux, as defined broadly by OIN. OIN's current license agreement generally provides that any patent licenses survive a change of control—which is to be expected for such a license.[33] (Novell, however, probably agreed to an earlier version.)

In addition, Novell had previously made a public Novell patent pledge:

> Novell will use its patent portfolio to protect itself against claims made against the Linux kernel or open source programs included in Novell's offerings, as dictated by the actions of others. … In the event of a patent claim against a Novell open source product, Novell would respond using the same measures generally used to defend proprietary software products accused of patent infringement. Among other things, Novell

[32] www.computerworld.com/article/2514328/enterprise-applications/attachmate-to-retain-novell-unix-copyrights.html.

[33] www.openinventionnetwork.com/joining-oin/oin-license-agreement/.

> would seek to address the claim by identifying prior art that could invalidate the patent; demonstrating that the product does not infringe the patent; redesigning the product to avoid infringement; or pursuing a license with the patent owner.[34]

This was couched as a covenant rather than a license. Thus, it was not clear whether the pledge would inure to Attachmate, and if so, whether Attachmate could withdraw it. Because the ownership of the patents and company diverged, it was unclear whether the patents remained in Novell's patent portfolio. Also, it was unclear whether the pledge would continue to encumber the rights of CPTN, as a license would have, or evaporate with the transfer, as a covenant might.

The DOJ reviewed the transaction and later issued a press release[35] about the modification of the Novell sale. The DOJ required Microsoft to "sell back to Attachmate all of the Novell patents that Microsoft would have otherwise acquired," leaving Microsoft only with a license. The DOJ also unwound the sale to EMC of 33 patents and patent applications related to virtualization. The DOJ's press release said, "All of the Novell patents will be acquired subject to the GNU General Public License, Version 2 ... and the Open Invention Network (OIN) License." It is unclear exactly what the release meant by the GPL reference, given that GPL 2 has no express patent grant (perhaps referring to an implied license), but the statement clarifies that the patents will remain subject to OIN license terms. The DOJ also said, "CPTN does not have the right to limit which of the patents, if any, are available under the OIN license," suggesting that the limiting elections possible under the OIN license cannot be made in the short term by CPTN. The full documents were not made public but may be available when the settlement is filed with the court as a part of the Tunney Act approval process. The Tunney Act is a US antitrust law that, among other things, requires that the government and an antitrust defendant disclose communications related to the settlement process.

[34] www.novell.com/company/policies/patent/.

[35] http://www.justice.gov/opa/pr/2011/April/11-at-491.html

H. Distribution (2010)

On November 17, 2010, the English High Court ruled in *Football Dataco Limited, et al. v Sportradar GmbH, et al.* that the act of making material available to the public by online transmission occurs where the transmission takes place, and not where it is received, for the purposes of copyright.

Plaintiffs alleged copyright infringement of their live streams of data (such as goals scored, goal scorers, penalties, yellow and red cards, and substitutions) for soccer matches in a package called "Football Live." The defendants were in the business of assembling similar data from public sources. The defendants' data were stored on web servers in Germany and Austria but could be accessed via links from other locations, including the UK.

The defendants argued that there was no jurisdiction for the English Court because no acts of infringement had occurred in the UK. The court held that Sportradar had not done any act of reproduction (in respect of copyright) or extraction (in respect of database rights) in the UK.

Database rights are governed by Article 7(2)(b) of the Database Directive: "Any form of **making available** to the public all or a substantial part of the contents of a database by the distribution of copies by renting, by on-line or other forms of transmission" [emphasis added]. The court stated that the question of where the act of *making available* occurred under this Directive was related to where *making available* occurred for the purpose of s.20 Copyright Designs and Patents Act 1988. The court analogized to the precedent of where a "broadcast" occurred under the Directive on Satellite Broadcasting and Cable Re-transmission for broadcasts originating within the EU; under that directive, the act of broadcast occurs where the signals are introduced under the control of the person making the broadcast into an uninterrupted chain of communication (which is referred to as *emission theory*).

The court stated, "[T]he act of making available to the public by online transmission is committed ... only where the transmission takes

place. It is true that the placing of data on a server in one state can make the data available to the public of another state but that does not mean that the party who has made the data available has committed the act of making available by transmission in the State of reception."

This decision tangentially effects the interpretation of copyleft obligations of open source licenses. The effect turns on the extent to which copyleft obligations are triggered by *making available* software as opposed to *distribution* of software. It has long been a source of speculation in open source licensing whether *distribution* in a license like GPL 2 could be interpreted to include *making available*—the former being primarily a US legal concept and the latter common in Europe and the UK. There is not settled law on whether SaaS use of software constitutes *making available*—though it would not constitute distribution in the US. Those who make code available but do not distribute it (such as via a SaaS product) have puzzled over whether they could inadvertently trip copyleft obligations if the software were accessed in Europe and the UK. Now, at least in the UK, there seems to be some comfort on the issue.

I. Arduino

Arduino is a well-known project promoting open hardware for devices designed to interact with their physical environment (such as robotics or motion sensors). A dispute about the Arduino trademark erupted between Arduino LLC and Arduino SRL in a Trademark Trial and Appeal (TTAB) case filed in late 2014, followed by a lawsuit in January 2015.

Arduino LLC, a US entity incorporated in 2008 by a group of founders, promulgated open software and hardware designs. Hardware devices based on these designs were manufactured by Smart Projects SRL, an Italian entity formed by one of the founders. Arduino LLC runs a certification program for third-party manufacturers that want to use the Arduino brand. Arduino LLC filed a US trademark application in April 2009, and a registration was issued in 2011.

Smart Projects filed a petition with the US Patent and Trademark Office (USPTO) in October 2014, asking that office to cancel Arduino

LLC's trademark on "Arduino." Smart Projects then changed its company's name to Arduino SRL. Arduino LLC responded by filing a trademark infringement lawsuit against Arduino SRL in US District Court, District of Massachusetts. In July, the TTAB suspended the trademark case pending the outcome of the ongoing civil proceedings.

J. WordPress Domain Names

The WordPress Foundation, operator of the WordPress open source software project, sued the owner of the WordPressHelpers.com domain name, alleging trademark infringement. In May 2015, the defendant brought an opposition proceeding at the TTAB, opposing registration of the WordPress trademark application. In early September, the TTAB administrative judge denied the defendant's motion for default judgment and suspended proceedings until the civil case was complete. The parties then settled out of court on September 21, 2015, with the defendant agreeing to cease use of any WordPress mark and to transfer the domain names to the WordPress Foundation.

K. Court Interprets "Commercial" Use Under Creative Commons License (2014)

A German court of appeal, in a 2014 dispute over what is "commercial" use of a photograph licensed under the Creative Commons Attribution Non-Commercial 2.0 license (CC-BY-NC), decided that the publication of a photograph on the Deutschlandradio website did not represent commercial use for the purposes of this license.

Deutschlandradio had used the photograph licensed under the CC-BY-NC to illustrate a story on its website. The photographer initiated proceedings in Germany for unlawful commercial use.

The court applied the license definition and not that of "commercial use" under German law, as the license is meant for worldwide use and the court held that the use did not amount to commercial use as defined by the CC-BY-NC license. Accordingly, the photographer was not entitled to a license fee.

However, Deutschlandradio had cropped the photographer's name out of the photograph used on the website. The court found this to be a violation of the license terms, so the plaintiff won an injunction against the defendant for use of the photograph in its cropped form.

L. Court Enforces Creative Commons License (2015)

In *Drauglis v. Kappa Map Group*,[36] a district court confirmed that a photo licensed under the Creative Commons By-SA 2.0 license could be used for commercial purposes. The plaintiff photographer had uploaded his photo to Flickr and made it available under CC-BY-SA. The defendant downloaded a copy of the photograph from the plaintiff's Flickr account and began publishing and selling a local atlas with that photograph on the cover. Nothing on the front cover of the atlas identifies who took the photograph, but text on the bottom of the back cover of the atlas identifies the photograph, the plaintiff, and the Creative Commons license. The front cover did have a copyright notice by the defendant, which the plaintiff challenged as misrepresenting the photograph as the defendant's own.

The CC ShareAlike licenses are roughly equivalent to copyleft licenses for software. In this case, the dispute focused on whether the defendant's use exceeded the scope of the license by violating attribution requirements. The court held that the defendant did not violate the terms or exceed the scope of the license under which the plaintiff had published the photograph. The court further held that the defendant's copyright notice on the first page of the atlas was not "conveyed in connection with" the photograph on the front cover, and the court therefore granted the defendant's motions for summary judgment on both counts.

[36] 128 F. Supp.3d 46 (D.D.C. 2015).

M. Artifex v. Hancom[37]

Artifex is the developer and licensor of Ghostscript, a popular PDF rendering library that is dual licensed under GPL and commercial licenses. Artifex filed suit against Hancom, a South Korean company that sold Hancom Office, a suite of productivity software. Hancom had incorporated Ghostscript into its word processing software product without complying with the terms of GPL.

The parties settled this case for an undisclosed amount before trial. But the case was notable because of several statements by the court in response to motions by the parties. Artifex had alleged breach of contract as well as copyright infringement, and on a motion to dismiss, the court refused to dismiss the contract claim. Hancom argued that no damages were due because the GPL license is granted free of charge, but the court disagreed, stating that under California contract law "the jury can use the value of the commercial license as a basis for any damages determination." The court also suggested that unjust enrichment was a potential basis for damages.

N. Patent Trolls

There have been a number of suits by nonpracticing entities that accuse open source software. These include, for instance, *IP Innovation LLC v. Red Hat Inc.* (filed October 9, 2007, E.D. Texas), *Software Tree LLC v. Red Hat Inc.* (filed March 3, 2009, E.D. Texas), and *Bedrock Computer Technologies, LLC v. Softlayer Technologies, Inc.* (filed June 16, 2009, E.D. Texas). This is by no means an exhaustive list; these cases are not unique to the open source context, as software patent troll cases abound in the Eastern District.

The most serious recently active patent troll is called Sound View Innovations, which filed over a dozen lawsuits in 2016-2019 targeting companies using open source software like Hadoop and JQuery. The underlying patents came from AT&T Bell Labs and Lucent

[37] *Artifex Software, Inc. v. Hancom, Inc.*, U.S. Dist. 2017 WL 1477373 (N.D. Cal. Apr. 25, 2017).

Technologies, and were purchased by Sound View from Alcatel Lucent in December 2013.[38]

In 2019, Rothschild Patent Imaging sued the Gnome Foundation for patent infringement, and the open source community, including OIN, promptly came to Gnome Foundation's aid to find prior art, seeking to invalidate the patents.

The contrast between the Sound View cases and the Rothschild case illustrates that it is not a good monetization strategy to sue open source projects: they don't have much money, they have a lot of friends with an incentive to challenge patents, and they generally cannot settle the way a private plaintiff would.

V. The Cast of Characters

Open source claims are not like other intellectual property claims. Treating them like ordinary intellectual property claims may lead an accused party to do exactly the wrong thing on a strategic level.

Open source claimants today mostly come in two varieties: the advocates and the strategists. Of course, free software has always had its zealous advocates—it is, after all, a movement started for the greater good, based on the idea that access to software source code should be a techno-political right. The first claims brought in the area of open source enforcement were claims brought by advocates. Those efforts bore the clear fingerprints of advocacy and focused on embarrassing the miscreants and demanding their compliance. This strategy meant that filing a lawsuit was a last resort, and most compliance disputes were resolved informally and without resort to legal process. Although the early enforcement actions of SFLC sometimes included monetary settlements, these were often characterized as a contribution to offset the cost of litigation—and rightly so, as compared to the average intellectual property dispute the amounts involved were minimal.

[38] This illustrates one of the tenets of the License on Transfer network: that most patent trolling results from sales of patents by practicing entities, such as when those entities are liquidated due to business distress, or sell off non-performing assets. https://lotnet.com/faq/.

Settlements were announced publicly in greater detail than is customary in intellectual property disputes.

Here are examples of this type of claimant:

- **Free Software Foundation (FSF). The** FSF stewards the GPL and runs the GNU project. Although it was originally involved in enforcement of GPL violations, most of its enforcement activities have now been taken over by SFLC.
- **Software Freedom Law Center (SFLC). This** is a legal advocacy organization that represents pro bono clients, including FSF, in GPL enforcement and other free software law matters. Today it conducts relatively little formal enforcement of licenses.
- **gpl-violations.org.** As described above, this organization has brought various enforcement actions in Europe.
- **Software Freedom Conservancy (SFC).** This organization is today's most active community enforcer. It has not yet filed any lawsuits, but it has conducted informal enforcement claims. It represents the project BusyBox, which is historically the subject of most GPL enforcement claims. It also spearheaded and funded the lawsuit brought by Hellwig against VMWare, as described in this chapter.
- Individual authors acting via any of the above.

Software Freedom Conservancy

Over the past few years, SFC has undertaken a growing role in the enforcement of GPL. SFC's venue of choice is Germany, and indeed, many open source plaintiffs seem to be gravitating toward Germany because it is a plaintiff-friendly venue. This phenomenon is not unique to open source claims—some have called Germany the "Eastern District of Texas for soft IP lawsuits."

In 2015, the Software Freedom Conservancy added several new member projects: QEMU, a generic machine emulator and virtualizer; The Bro Project, a network traffic analysis framework and security platform; and Godot Engine, a 2-D and 3-D cross-platform game

engine. Although these are relatively small projects, SFC continues to grow its list of member projects and, presumably, will ultimately be involved in any enforcement actions for those projects. However, SFC activity seems to be slower than it once was. The SFC's involvement in the *Hellwig v. VMWare* case was quite unpopular in the technology industry and may have compromised its funding efforts.

Partially in response to the McHardy trolling cases, Software Freedom Conservancy, along with FSF, published a document called "The Principles of Community-Oriented GPL Enforcement" in October 2015.[39] These guidelines are meant to enunciate a consistent and common approach to enforcement taken on behalf of the community, as opposed to enforcement taken for private gain. Principles enumerated in the guidelines include prioritizing software freedom over all other ancillary goals, using legal action only as a last resort, and offering flexibility on rights restoration under GPL 2's termination clause (GPLv2§4).

Strategic Lawsuits

The 2010s saw the emergence of a new kind of claimant—the strategic claimant—who has most of the same goals as other intellectual property claimants and who sometimes is both a copyright and a patent claimant. Strategic claimants are building on the foundation of *Jacobsen v. Katzer* to enforce their rights in open source software via intellectual property disputes. And like other intellectual property disputes prosecuted by strategic plaintiffs, such disputes tend to be the battles of titans. This kind of plaintiff wants to gain a strategic edge, whether via delaying a competing product's release, embarrassing a competitor, or extracting money to de-fund a competitor's development plans. Notably, however, this kind of claimant wants damages or remedies for past infringement and not merely compliance going forward. Settlements tend to be slower and more confidential.

[39] http://sfconservancy.org/copyleft-compliance/principles.html

In between the advocates and the strategists are individual authors who are not affiliated with an advocacy organization. They often are not represented by counsel at all—eschewing the pure political motivation of the advocates and therefore not sharing their goals. These authors seldom make formal claims and may demand modest license fees to grant an alternative license that would obviate the need to comply with open source requirements. (Advocates, on the other hand, would usually refuse to agree to other licenses.) Obviously, the claims of these individuals are best handled via a quick and confidential settlement. Attorneys handling such claims must be mindful of ethical issues relating to interaction with unrepresented parties[40] and are best advised to keep the discussion straightforward and the documents short. Adding complex releases, indemnities, and patent grants will usually backfire.

The possibility still exists that such authors will become "copyright trolls." As open source software becomes ever more pervasive and some companies still have inadequate controls to avoid unintended use of open source, it is quite possible that a popular bit of code could become the subject of "submarine" copyright claims.

VI. Running the Numbers

Since copyleft enforcement began in the late 1990s,[41] there has been a steady increase in compliance actions. Because most enforcement starts as an informal dialogue,[42] statistics on the number of enforcement actions and their yield in settlements or damages can be hard to verify.

[40] Of course, the ethical rule concerns represented parties. (See ABA Model Rule 4.2.) However, unrepresented parties can easily engage representation, so lawyers negotiating with unrepresented individuals should be sensitive to the potential need to avoid direct communication with a claimant.

[41] See Bradley Kuhn, "13 Years of Copyleft License Compliance: A Historical Perspective," October 12, 2012, www.ebb.org/bkuhn/talks/Open-World-Forum-2012/Compliance-History/compliance.html.

[42] Bradley Kuhn mentioned that "99.999% of GPL enforcement matters get resolved without a lawsuit," February 1, 2012, http://sfconservancy.org/blog/?tag=gpl.

According to the Executive Director of the Software Freedom Conservancy (SFC), Bradley Kuhn, SFC logged 100 GPL violations as of 2007, and as of 2012, it was pursuing (or planned to pursue) over 300 violations, with new reports coming in each week.[43] Because copyleft advocacy groups focus mostly on compliance and only request monetary compensation for legal work and related expenses, the "damages" paid for these actions are not terribly significant: SFLC reported its highest revenue from enforcement of $204,750 for fiscal year 2010. As a 501(c)(3) not-for-profit entity, SFC files Form 990 with the IRS and CHAR-500 with New York State, and makes its filings available on its web site.[44] Those filings indicate that SFC has not received any enforcement revenue since 2012.

Harald Welte and the gpl-violations.org project did not publish such numbers (only keeping track of reported GPL violations in the project's internal request-tracking system), but their legal archive contained discussion threads that listed at least a few publicly discussed potential violations each month. After its start in 2003, the project "hit the magic '100 cases finished' mark in June 2006, at an exciting '100% legal success' mark." In 2008, the project reported, "In the past 30 months, gpl-violations.org has helped uncover and negotiate more than 100 GPL violations and has obtained numerous out-of-court settlement agreements." In addition to information about court cases pursued in Germany and France, gpl-violations.org claims that in virtually every instance where a violation has been found and action taken, the enforcement has been successful—including out-of-court settlements with a number of large vendors, such as Fujitsu-Siemens. The project published a separate Frequently Asked Questions (FAQ) page aimed specifically at source code releases, which was "compiled as a result of more than sixty successful GPL enforcements."[45]

[43] See Bradley Kuhn, "13 Years of Copyleft License Compliance: A Historical Perspective," October 12, 2012, www.ebb.org/bkuhn/talks/Open-World-Forum-2012/Compliance-History/compliance.html.

[44] https://sfconservancy.org/about/filings/.

[45] Unfortunately, the website for gpl-violations.org is no longer available; however, these facts were taken from that site in 2011, when it was still online.

Many potential open source users ask about the likelihood that a disclosure of proprietary code will be forced due to the "viral" effect of GPL.[46] Because most enforcement has focused on failure to include licensing notices or source code offers for third-party GPL code, the question of the GPL's scope—and the concomitant question of the necessity to release derivative works containing proprietary code—has not been the subject of most enforcement actions so far. (For an exception, see the *Jin v. IChessU* case and the *Hellwig* case, discussed above).

The most widely publicized release of source code as the result of a claimed GPL violation was the one made by Cisco in connection with the early days of the Linksys dispute (discussed in II.C above). Bradley Kuhn cites the NeXT Objective C compiler as the first GPL violation, in 1989. As a result, NeXT released its source code for the compiler.[47] Asus recently released changes to the Linux kernel in response to demands from the open source community.[48] In August 2013, Samsung made a code release after an anonymous hacker (a 19-year-old college student based in Europe) posted to GITHUB what she claimed were snippets of Linux code being used by Samsung in its Android exFAT functionality.[49] Although the hacker clearly had no right to post the code, Samsung worked proactively with SFC to make a source code release.[50] These are only examples, and necessarily anecdotal. Releases of formerly proprietary source code tend to be done with a minimum of fanfare and are likely to go unremarked unless the release is made after

[46] In fact, this generally represents a misunderstanding of the nature of open source enforcement and the available remedies for violation of licenses. For more discussion, see Heather Meeker, "Open Source and the Eradication of Viruses," March 19, 2013, *Open Source Delivers*, http://osdelivers.blackducksoftware.com/2013/03/19/open-source-and-the-eradication-of-viruses/.

[47] www.ebb.org/bkuhn/talks/Open-World-Forum-2012/Compliance-History/compliance.html.

[48] Ryan Paul, "Asus Resolves Eee GPL Violation, Releases asus_acpi code changes," *Ars Technica.*, http://arstechnica.com/information-technology/2007/11/asus-resolves-eee-gpl-violation-releases-asus-acpi-code-changes/

[49] See "Busted for Dodging Linux License, Samsung Makes Nice with Free Code," *Wired,* August 20, 2013, www.wired.com/wiredenterprise/2013/08/samsung_exfat/.

[50] http://sfconservancy.org/news/2013/aug/16/exfat-samsung/

a copyleft violation has been widely publicized. Many enterprises make regular and automatic releases for their products' open source components, so a source code release seldom appears unusual or remarkable.

Some distributors of third-party code have decided to pull non-compliant products from their virtual shelves in answer to third-party GPL violation complaints, as Apple did in 2010 with GNU Go[51] or in 2011 with VLC Media Player.[52] This represents a kind of "hidden enforcement" that does not reflect a dialogue between author and infringer at all, taking place via the DMCA notice and takedown of a third party. If the alleged infringer is a small company, the removal of its product from an important reseller store may be the end of its business. As with any DMCA takedown request, the reseller has no direct legal obligation to resolve the dispute on its merits, and so it is up to the reseller how much investigation and analysis to pursue.

VII. What to Do If You Receive a Claim

Best practices include, immediately and foremost, an assessment of the type of claimant facing you. Advocates primarily want compliance and tend to be tolerant of foot faults.[53] A sincere, timely, and robust compliance program will often meet their demands. But strategic plaintiffs can draw you into a world of complex and unsettled open source law, in addition to the typically painful process of intellectual property litigation.

If you receive an open source claim, it is crucial to act on it quickly. That may sound simple, but it can be unexpectedly difficult because not all open source claims are made through formal channels. Companies should train their engineering and support staff to recognize

[51] See, for example, Brett Smith, "GPL Enforcement in Apple's App Store," May 25, 2010, www.fsf.org/news/2010-05-app-store-compliance/.

[52] For details on this and the controversy over GPL versus App store terms, see http://heathermeeker.squarespace.com/news/2011/1/9/gpl-apps-pulled-from-iphone-store.html.

[53] Bradley Kuhn, often considered the informal czar of GPL compliance, discusses this here: www.ebb.org/bkuhn/blog/2009/11/08/gpl-enforcement.html.

complaints and take them seriously. Complaints often come in the form of emails sent to technical employees who, typically, represent the company's face to the software development community. Claimants may not be aware that there are accepted formal means of making claims, such as service of process. In contrast, a traditional intellectual property complaint will often come to a legal or management representative, who will easily recognize it as a legal claim, or via a formal complaint. In addition, technical employees should be trained to react immediately to any request for source code made pursuant to a copyleft license like GPL. If that request is not answered on a timely basis, or if the company demands fees to fulfill it, a formal open source claim is likely to follow quickly. Of course, only a downstream recipient has the right to make such a request, and only the upstream author has the right to make a legal claim.[54] So in truth, these are not infringement claims. However, recipients whose demands for source code are unmet often complain to those who do have the right to bring a claim, or advocacy organizations that will take up the matter on their behalf.

In truth, claims from advocates are the easiest to handle. The goals and actions of advocates are predictable. The emphasis is on compliance and attribution, and only secondarily on damages. Injunctive action is usually not on the table at all. Contrast this approach with that of strategists, who would generally prefer injunction and damages over compliance because such rulings serve their goal to disrupt the market for a competitive product.

Strategic plaintiffs may also make multiple claims, only one of which will be copyright infringement. The Federal Circuit's two reversals of defendant wins in *Oracle America v. Google* has resulted in a common litigation strategy to add patent claims to copyright claims in order to skew appeals jurisdiction toward an intellectual property plaintiff friendly court. As a baseline, any lawsuit that involves open source licensing is likely to touch upon novel issues of law, as well as

[54] The conditions in a license like GPL run downstream only. Recipients are not third-party beneficiaries of the GPL; those who take the position that the GPL is not a contract would say that there can be no third-party beneficiary under a conditional license.

draw significant attention from the press and the free software community.

VIII. Best Practices

In summary, here are points to remember about open source litigation and disputes.

- Take open source claims seriously, even if they are made informally or by unrepresented parties. Train your technical staff to recognize claims.
- If you receive a claim, act in a timely fashion. Ignoring open source advocates who complain about your practices, formally or informally, is not an effective strategy.
- Don't treat open source claims like other IP claims. Most open source disputes can be resolved more quickly and economically than other intellectual property disputes.

Obstacles to Enforcement[55]

Before GPL was ever enforced in court, various commentators postulated many barriers to enforcement, most of which have been discredited. In its lawsuit against IBM, SCO claimed that the GPL violated the US Constitution.[56] A later suit claimed that the GPL violated antitrust law.[57] Many said the license was unenforceable because it had never been tested in court—a solipsistic and uninformed argument that, fortunately, is seldom heard anymore; those making it probably did not consider how deeply it would contradict fundamental legal principles to require that a court hold a contract enforceable in advance—a rule that would wipe out freedom of contract in a single

[55] For an excellent article on the topic, see Jason Wacha, *Taking the Case: Is the GPL Enforceable?* reprinted at http://digitalcommons.law.scu.edu/chtlj/vol21/iss2/5/. Wacha's article discusses the enforcement challenges typically cited in the 2000s.

[56] *Open Letter on Copyrights*, Daryl McBride, The SCO Group Inc., December 4, 2003, www.sco.com/copyright/.

[57] Order dismissing *Wallace v. Free Software Foundation, Inc.* (S. Dist. Ind., October 28, 2005).

blow. Contract formation defenses may linger, but they are strategically unattractive to a potential licensee who has no other way to exercise distribution rights (see Chapter 6: What Is Distribution?). Unsurprisingly, none of these arguments has been remotely successful in an actual lawsuit.

However, there are some obstacles to enforcement worth considering. This is not to imply that one should not enforce open source licenses; in fact, it can be quick and effective to enforce them. But before you make the choice to send a demand letter or file a lawsuit, you should make sure you have not missed one of these potential defenses.

Don't Commit Intellectual Property Suicide

If you are going to throw the stones of noncompliance at open source licenses, be sure you are not living in a glass house. Before you bring a claim, you should be sure your own open source compliance is in order, at least to the extent that you have not infringed the rights of the defendant you are about to sue.

Counterexamples to this rule of thumb include *Jacobsen v. Katzer* (discussed in Chapter 14: Open Source and Patent Litigation Strategy) and the Versata/Ameriprise/Ximpleware cases (now settled and dismissed).

Joint Authorship, Standing, and Joinder

Open source is a collaborative development model. Many open source projects emphasize that they aim to create a single, cohesive work of authorship from the work of many contributors. But then, who is the author of the copyrightable work that consists of the software?

In the United States, authors who collaborate on a work can be considered joint authors. Joint authors need not be conjoined in time or geography.[58] The copyright law states that a joint work arises when both

[58] See for example, *Edward B. Marks Music Corp. v. Jerry Vogel Music Co.*, 140 F.2d 267 (2d Cir. 1944).

authors intended, at the time the work was created, "that their contributions be merged into inseparable or interdependent parts of a unitary whole." The Second Circuit has stated that "Parts of a unitary whole are 'inseparable' when they have little or no independent meaning standing alone. ... By contrast, parts of a unitary whole are 'interdependent' when they have some meaning standing alone but achieve their primary significance because of their combined effect."[59] Whether a work is a joint work may turn on the intent of the authors. Nimmer observes, "The distinction lies in the intent of each contributing author at the time his contribution is written. If his work is written 'with the intention that [his] contribution ... be merged into inseparable or interdependent parts of a unitary whole,' then the merger of his contribution with that of others creates a joint work. If such intention occurs only after the work has been written, then the merger results in a derivative or collective work."[60] Opinion is split on whether the individual authors need contribute copyrightable expression or whether contribution of ideas alone will suffice.[61] Contributors to an open source project may find it difficult to argue that their work is not a joint work.

The default rule for a jointly owned work is that each author has the right to use the work without limitation, subject to default law obligations to account to other authors.[62] Clearly, such rules—which were developed to apply to works like books, music, and audiovisual works—were never intended to address the kind of massive collaboration that arises from modern technology like GITHUB, Wikis, and the like.

Because all authors can grant any nonexclusive licenses they like, if one author sues for copyright infringement, courts often require all authors to be joined as parties to the suit. Under 17 USC Section 501(b), a court "may require the joinder, and shall permit the intervention, of

[59] *Childress v. Taylor*, 945 F.2d 500, 505 (2nd Cir. 1991).
[60] *Nimmer on Copyright* Section 6.05 (2005).
[61] *Ashton-Tate Corp. v. Ross*, 916 F.2d 516 (9th Cir. 1990).
[62] *Oddo v. Ries*, 743 F.2d 630 (9th Cir.1984).

any person having or claiming an interest in the copyright." Otherwise, the defendant would be able to claim it had a legitimate license from one of the authors who was not a party.

Moreover, under US law, only the author or copyright owner has standing to bring a lawsuit for infringement. Although any author could sue for infringement of his own portion of the project that was considered its own work of authorship, a court would require, if equity demands it, that the principal owners of the infringed material be joined to any case meant to enforce rights in the work. The court has discretion to require combined or separate lawsuits.[63]

The risk of these issues arising has increased in recent years as some open source legal experts have suggested joint authorship as a remedy to open source trolling.[64]

[63] *Edward B. Marks Music Corp. v. Jerry Vogel Music Co.,* 140 F.2d 268 (2d Cir. 1944) held that co-owners were not indispensable parties; for the opposite view, see *Key West Hand Fabrics v. Serbin, Inc.*, 244 F. Supp. 287 (S.D.Fla. 1965). Case law on the subject is scant.

[64] Chestek, Pamela S, A Theory of Joint Authorship for Free and Open Source Software Projects (July 2, 2017). Available at SSRN: https://ssrn.com/abstract=2999185 or http://dx.doi.org/10.2139/ssrn.2999185.

Chapter 20

Open Standards and Open Source

Much has been written about open standards, which are sometimes confused with open source software licensing. Open source licensing is a paradigm specific to software; *open standards* is a more general term with no single accepted definition. However, open standards mostly refer to standards that are developed with no known patent claims or with an understanding that any patent claims will be licensed on a royalty-free and nondiscriminatory basis.

What Is a Standard?

Standards are specifications that many in an industry agree to use in order to maximize the interoperability of products and promote commerce. Consider the quintessential example of nuts and bolts. Hundreds of years ago, every nut and bolt pair had its own thread pattern. As time went on, nuts and bolts were mass-produced and standardized so that any nut and bolt would be made for a particular thread configuration,[1] and one could buy nuts and bolts with those specifications and know they would work together.

Particularly in the United States, standardization takes place mostly by consensus. So it is possible that an industry can lack critical mass for standardization. Also, there can arise competing standards—as in the VHS and Betamax wars. Lack of standardization makes consumers and producers cautious about buying products, which decreases economic activity.

[1] For a great explanation of threading standardization, see the Wikipedia article http://en.wikipedia.org/wiki/Screw_thread.

Standards-setting organizations (SSOs), such as IEEE and W3C, promote the orderly development of standards. If industry players decide to work on a standard, they usually create a working group under the aegis of an SSO. Participants in the *working group* discuss the content of the standard. They don't always agree, but when they do, the standard is adopted and published. To clarify what patents cover the standard, SSOs usually have intellectual property policies that participants must agree to prior to joining a working group. Obviously, it is possible that not all patent holders might participate in the working group, and if that happens, the standard may be created and adopted but may be thwarted by third-party patent claims. In the optimal case, all of the important patent holders participate in the working group and agree to license their patents to those who want to practice the standard.

Beyond this, the rules can vary a great deal. Some standards are royalty-bearing but licensed on *reasonable and nondiscriminatory* (RAND) terms. Some are licensed on RAND terms with no royalties, sometimes called RAND-z. Sometimes the standard is intended to be patent-free, and that is usually accomplished by the working group's taking care to avoid including anything in the specification that is likely to be subject to patent claims.

One reason for the popularity of RAND licensing is that it deflects antitrust liability. RAND means no one can be excluded from practicing the standards, and licenses must be offered to everyone—though royalties can be adjusted for volume, use case, and so forth.[2] When standards are mandated by government regulation, RAND can be obligatory, but this is more common in Europe than in the United States.

[2] The Department of Justice Guidelines generally consider intellectual property licensing to be pro-competitive. See Antitrust Guidelines for the Licensing of Intellectual Property, April 6, 1995, Department of Justice, www.usdoj.gov/atr/public/guidelines/0558.htm. However, license arrangements can stray into the prohibited area of horizontal output restraints or price fixing. Famously, Rambus Inc. was subjected to a protracted legal battle over the failure to disclose intellectual property related to a standard in the course of a standards development process; see www.ftc.gov/os/adjpro/d9302/060802commissionopinion.pdf.

Each SSO has its own rules for licensing of standards-essential patents. For instance, IEEE requires RAND licenses,[3] W3C has RAND-z licenses,[4] and IETF requires RAND licenses.[5]

Standards and Open Source

Open source software can be a great vehicle for standards adoption. It brings down the licensing barriers for the use and adaptation of reference software, and particularly if it is offered under a permissive license, it promotes widespread adoption. But clearly, if open source software implements a royalty-bearing or *non-free* standard, notwithstanding the royalty-free grant of copyright under the open source license, users would be at risk of patent infringement claims when using the software.

While open source licenses grant patent rights, they only grant rights in patents owned by the author of the software. It is possible—in fact, common—that software can practice inventions claimed by patents owned by third parties. When practicing a standard, this is almost surely the case.

For example, FFMPEG is a very popular open source audiovisual codec, which encodes and decodes data files stored in the various standard formats. FFMPEG is licensed under open source licenses, but the patents covering certain MPEG standards are owned by others. Anyone seeking to use the FFMPEG software must consider not only the need to comply with the open source licenses that govern its use but also the need to get a separate patent license—or of course, take a risk of patent infringement claims. This is an example of how open source software licensing and non-free standards can clash.

[3] http://standards.ieee.org/faqs/copyrightFAQ.html.

[4] www.w3.org/Consortium/Patent-Policy-20040205.

[5] ftp://ftp.rfc-editor.org/in-notes/bcp/bcp79.txt.

Different Rules

Standards licensing has been around for a long time, and although it can be complex, it is settled and familiar territory to those who deal with patent issues outside of the open source context. So in a way, it is easier for many companies to become comfortable with standards licensing than open source licensing. When the two interact, it can result in a clash of cultures and skills sets. Standards licensing is mostly the purview of patent lawyers, and open source licensing is more familiar to corporate lawyers and businesspeople.

In some ways, they are similar. Open source licenses create a patent *commons* to practice the copyright in the software, and standards licensing also creates a patent commons to practice a standard. But beyond that, the two are very different.[6]

First, they have different boundaries. A standard is usually expressed in a copyrightable document called a specification, but generally, the copyright is not the valuable part of the document. The more important rights are the patent rights that cover the standards. These are sometimes called *standards essential patents* (SEPs), and the claims of those patents are often called *necessary claims*. The claims that are licensed are those necessary to practice the standard.

Open source licenses, in contrast, are bounded by the software they cover. When they include patent license grants, they often use similar terminology such as *essential patent claims*. But it is easier to tell if open source software practices a patent than to determine whether a patent is necessary to practice a standard. That is because the standard describes a technological solution but does not instantiate or implement it. A single standard can have many implementations; software is a single work.

The rules are also quite different. Standards licenses often extend only to those who practice the standard exactly as adopted and that

[6] For in-depth information on standards and the implementation of the intellectual property rules of standards bodies, see Jorge L. Contreras (Ed.), *Technical Standards Patent Policy Manual*, American Bar Association Publishing (2007).

practice it fully. Open source licensing places no limits on modification of the subject matter it licenses. But it is worth noting that the patent licenses in neither paradigm extend to downstream modification. The difference is that while open source licensing encourages divergence, standards licensing encourages uniformity.

Also, open source licenses do not require licensees to grant back rights in their own patents or agree to a variety of other terms that sometimes attend standards licensing. There is no one set of standard terms for a standards license, and in RAND or RANDz schemes, each licensor may offer its own set of terms. Sometimes, licensors band together for "one-stop shopping" in RAND patent licenses, such as in the case of MPEG-LA—but that is the exception, not the rule.

Chapter 21

Open Hardware and Data

As the idea of *open* has matured, there have been attempts to expand the model of copyleft outside the software context. Some have fared well: Creative Commons has ably applied copyleft (or, in the Creative Commons lingo, *ShareAlike*) licensing to non-software copyrightable works such as music, text, and audiovisual works. However, attempts to apply copyleft licensing outside of the context of copyrightable works have been mostly disappointing. This is not the fault of those who have attempted it; moving beyond copyright removes one of the pillars of copyleft, and without it, licenses are set adrift in an intellectual property limbo.

For copyrightable works, the threshold for protection is very low compared to the degree of creativity and expression usually embodied in a work. For example, any software program with more than a few lines of code probably qualifies for copyright protection. Other intellectual property subject matter, such as patentable inventions and rights in data, do not enjoy this level of protection. When applying copyleft to hardware, there is a threshold question of what is the quid pro quo for the copyleft conditions of the license. For software, the price of exercising the rights of copyright is adhering to the copyleft conditions. Any licensee trying to avoid the copyleft conditions is between a rock and a hard place. If one wishes to distribute software, one needs a copyright license to do so; no license means no rights.

Translating this to hardware is difficult in at least two ways: (1) establishing a threshold level of rights to enforce the copyleft conditions and (2) determining the materials necessary to enable reproduction of the design.

Most hardware designs are embodied in a specification or design document. Those making hardware are not particularly interested in distributing copies of product specifications. Merely "using" a specification, in the sense of reading it and following its instructions, is not an activity regulated by copyright. Even if a manufacturer violated the copyright in the specification by reproducing or distributing it, the damages for doing so would likely be trivial. Ideas are not protectable under copyright. The damages for exercising the copyright interest in the specification would have to accrue from the value of the work of authorship consisting of the specification, not the value of making the products. Moreover, the copyright protection of a specification may be thin to nonexistent. Copyright does not protect facts. Specifications consisting of tables of numbers, for example, might not be protected at all. Therefore, if a license says, "I will grant you the right to practice the copyright in this specification as long as you share your changes to it," the licensee might very well conclude there is nothing to be granted, refuse to adhere to the condition, and take a calculated risk that the licensor has no means to enforce the conditions.

Moreover, there is the question of what copyleft requires the licensee to disclose. The main copyleft condition in a license like GPL is making available the source code for the software. GPL 3 defines the source code for a program as "the preferred form of the work for making modifications to it." Although reasonable people might differ as to what *source code* means, fundamentally, it is the code that, when run through the correct compiler and builder, produces the object code being distributed. But what is the "source code" for hardware? To know how to make hardware, at a minimum, one needs a product design or specification. But this "source code" may be technology specific. That which is necessary to make a semiconductor is not the same as that which is necessary to make a television, a mobile phone, or a server blade. Using a general definition like "the designs necessary to make the product" is too broad and vague—it could require process technology that is expensive and not part of the product or commodity parts that no product manufacturer would even need to make for itself.

Finally, determining what is a *derivative* or modification that is subject to the copyleft terms of the license is hard enough for software; for hardware, it is even more difficult. The notion of a derivative work is confined to copyrightable works of authorship. Modifications to products by downstream licensees might correspond to changes in a specification, or not. They might be separately protectable under copyright law as changes to the specification or under patent law as new inventions. If they are new inventions, they may be changes to the original design or additional features, and the line of distinction is difficult to draw. A downstream licensee who makes a change may or may not have any rights in such a modification to license to others.

For these reasons, structuring hardware licensing by hanging one's hat on the practice of a copyright is risky. And of course, fundamentally, it is the power of patent, and not copyright, that protects hardware. What "open hardware" truly requires is a compulsion to license patents. Some licenses condition the practice of the rights in the specification on a requirement to license patents. But even this notion is vulnerable to a claim that the specification is not protectable in the first place, either because the specification is not protectable under copyright or because the creators of the specification have no patent rights to grant.

There is a time-honored way of creating a patent commons for hardware, but it is standards licensing rather than open source licensing. Standards licensing, however, is not usually an "open" model, and it is certainly not the kind of self-executing paradigm of cooperative development that is copyleft licensing; it is top-down licensing, with rights flowing in one direction from the specification's authors down to those practicing the specification. It does not account for licensing of downstream changes to the specification because traditional standards licensing is intended to discourage changes to the standard. In fact, in standards licensing, any modification of the standard is usually not licensed at all; some standards licenses require licensees to promise not to make alterations to products that make them inconsistent with the standard.

Therefore, open hardware licenses seem destined to walk the precarious line between standards licensing and open source licensing.

Although no copyleft hardware license has yet gained significant traction, it is early days, and this model may very well be developed and refined over time.

Open Data

Copyleft open data licensing has fared no better. The Open Data Commons Open Database License (OdBL) is a case in point. OdBL applies to Open Street Maps, a very popular database of mapping information, and licensees find it difficult to distinguish a "Derivative Database" from a new and separate database or a Collective Database. This kind of problem is endemic to any copyleft license, but it is particularly difficult for licenses without broad adoption. Unlike GPL, for which industry practice as to its scope was ironed out over many years, the OdBL is not widely used.

But part of the difficulty lies in the lack of clarity of the underlying law. In the United States, databases enjoy only very thin protection under copyright law. After the seminal *Feist* case,[1] that protection has been considered easy to avoid absent a wholesale copying of the data. The European Union has a more specific law on database protection—Directive 96/9/EC of the European Parliament and of the Council of 11 March 1996 on the legal protection of databases—but that law has been subject to little interpretation in the courts. Neither legal regime provides much useful guidance on what kind of changes or additions to a database might be derivative works, rather than new works.

Permissive Licensing

The conundrum of applying copyleft to hardware and data licensing does not, of course, apply in the same way to permissive licensing. Even permissive licensing faces the quid pro quo problem because the requirement to apply licensing or attribution notices depends on the granting of a right. Even so, there is much to be said for

[1] *Feist Pubs., Inc. v. Rural Tel. Svc. Co., Inc.*, 499 US 340 (1991).

simply placing databases and hardware specifications in the public domain and promoting cooperation and sharing by means other than licensing. An attribution requirement may seem innocuous until a single database or specification requires scores or hundreds of them.[2]

Examples

The following licenses have attempted to tackle the implementation of copyleft in non-software areas, but none has become widely adopted.

- CERN Open Hardware License. www.ohwr.org/projects/cernohl/wiki
- TAPR Open Hardware License. www.tapr.org/ohl.html
- Open Compute Project Hardware License. www.opencompute.org/blog/request-for-comment-ocp-hardware-license-agreement
- Tidepool Open Access to Health Data Software License. developer.tidepool.io/tidepool-license
- Open Data Commons Open Database License (ODbL). opendatacommons.org/licenses/odbl

[2] For an excellent real-world example, see http://peterdesmet.com/posts/illegal-bullfrogs.html. Thanks to Luis Villa for pointing out this rather priceless example of attribution run amok.

Chapter 22

Recent Developments: Commercial Open Source, Source Available Licensing, and Ethos Licensing

2018 was a watershed year for open source in business. In that year, many large "exits"—IPOs or corporate sales—occurred for commercial open source companies. Concurrently, a cascade of commercial companies migrated their business models to a hybrid model called "open core." None of this was new, but these trends achieved a critical mass in 2018 that changed the landscape of the software business. The first was a business trend, and the second was a licensing trend, and the two went hand in hand.

The Notable Deals[1]

In a flurry of deals in 2017-2018:

- MongoDB went public at a $1.6 billion valuation
- Salesforce acquired Mulesoft for $6.5 billion
- Pivotal went public at a $4 billion valuation
- Microsoft bought GITHUB for $7.5 billion
- IBM bought Red Hat for $34 billion
- Elastic went public at a $5 billion valuation

[1] For updated information, see the Commercial Open Source Software Company Index (COSSCI) maintained by Joseph Jacks. The numbers above are as reported, some are not official, and some are approximate. IPO valuations are typically imputed shortly after the initial offering, and do not usually equal the funds raised by the company.

- VMware purchased Heptio for a reported $500 million

These transactions demonstrated that commercial open source companies could make significant money. They also raised questions about the relationship of open source development to private capital, and what exactly it meant to be a commercial open source company.

Many open source advocates contend that private capital should not be used to develop open source software.[2] In fact, many popular open source projects have been spin-offs of private development, including Firefox (Netscape), Java (Sun), REACT (Facebook), Redis (Linkedin), and Kubernetes (Google). But advocates still contend that private companies are not the right stewards for open source development, being too much like the cathedral instead of the bazaar. Others see private capital as a powerful tool for funding open source development, particularly in a time when the sustainability of open source projects is in question. While huge projects like Linux and Kubernetes have plenty of development resources, many projects are starved of resources and "orphaned" in ways that cause them to become stale or even insecure. This is a philosophical dispute, and it will undoubtedly continue without a clear resolution.

But that begs the question: exactly what is a commercial open source company? For example, all of the companies that birthed the popular projects listed above would not be viewed as open source companies, even though they contribute heavily to its development. The difference is not about licensing, but about the nature of the company's philosophy and objectives. In other words, a company is an open source company only where "if the core OSS project did not exist, the company would not exist."[3] All of the companies mentioned in the notable deals above answer this definition. But all of them also had

[2] See for example https://lwn.net/Articles/786068/.

[3] See the comments of Joseph Jacks in https://medium.com/open-consensus/9-coss-company-value-creation-and-capture-fundamentals-9f0689c32ab0. Note that Jacks is my partner in Open Source Software Capital.

business models that provided their customers with value beyond the open source software at the core of their business.

What Color Are Your Razor Blades?

Many people wonder how it is possible to make a business centered around open source software. Perhaps the better question today is how not to do so. Open source software is like a public good, and public goods enhance business activity. The trick to running an open source business, and for that matter, running a prosperous society, is setting the mix between what is free and public, and what is proprietary. Free goods are like roads and proprietary goods are like bridges. In classic economic analysis, all roads should be private toll roads, because then only those who use them would have to pay, and they would not decay as a result of the tragedy of the commons. But that analysis leaves out two things: first, people hate toll roads, in large part because the act of paying the toll creates significant inconvenience (a transactions cost, in economic terms), and second, toll roads discourage the movement of people and goods, resulting in decreased macroeconomic activity. If everything is a free good, people will overuse it without contributing to its upkeep, but if everything is a private good, innovation is nearly impossible. Similarly, running an open source business is about setting the right mix of open source software development and capturing of private value.

Private companies that make businesses from open source usually cannot do so from licensing the open source software. But there are a number of business models that have worked well to monetize open source development. In the business strategy made famous by King C. Gillette, companies can build consumer bases by selling a product (such as a razor) cheaply and selling disposable accessories for it (such as razor blades) at a profit. In the famous book *What Color is Your Parachute?* the author asked job seekers to imagine the means to their next goal in concrete terms. For open source businesses, a concrete and clear business model is essential. It is the difference between a project and a business.

- **Support and Maintenance.** The most obvious model is support and maintenance. This model works reasonably well on a small scale, such as making an open source project into a full-time job for its developer. It is difficult to scale, however. Service businesses are notoriously low-margin because they rely heavily on human resources. The exception that proves this rule is Red Hat, which maintains the world's most popular Linux distribution. Red Hat does not sell licenses, but even its service revenue is heavily dependent on the quality control its brand represents. Accordingly, it maintains strong trademark policies. The conventional wisdom is that there is only one Red Hat. It turns on the world's most successful open source project—Linux—and drafts on huge economies of scale.
- **Dual Licensing.** Years ago, a business model called "dual licensing" was pioneered by a company called MySQL AB.[4] That database software was made available under GPL.[5] Therefore, those who used the software in proprietary applications could not distribute it without an alternative license. This practice of "selling exceptions" was initially supported by Richard Stallman, but he later viewed the practice more negatively.[6] In any case, it stopped working as a successful business model when the march of software to the cloud meant that it was much easier to comply with GPL, by merely offering the proprietary applications as SaaS instead of distributed products. While the dual licensing business model is not entirely gone, it faded after a 5-10 year period of popularity. Dual licensing is almost always deployed using GPL or AGPL, because they provide the strongest incentives for proprietary licensors to seek alternative licenses.

[4] Now owned by Oracle Corporation.

[5] Actually, it was a variant of GPL called GPL+FLOSS exception. But that made no difference to the business case for using the software in distributed proprietary applications.

[6] https://www.fsf.org/blogs/rms/selling-exceptions.

- **Open Core.** The open core model is the most successful open source business model for distributed software. It uses various "buckets" of software: an open core, usually licensed under a permissive license and often developed with a robust community, and one or two other buckets, licensed under source available or binary licenses. Where the core is maintained by the business,[7] the open core may be called a community edition, and additional bells and whistles are added to form an enterprise edition. This model is sometimes referred to as Freemium, but that is not quite the same. Freemium is more similar to what Richard Stallman terms "crippleware"—software that does not function fully without proprietary elements. The open core of an open core model is fully functioning software, but often omits the bells and whistles for deployment at scale in a large enterprise.
- **Embargo**. In this model, software is released first under proprietary licenses, and then under open source licenses. Paying customers get early access to the latest features. This model works particularly well with software for consumer electronics, where time-to-market is critical and stakes are high. This model is, to some extent, self-implementing in the MariaDB Business Source License.[8]
- **SaaS**. This model has historically worked well for software aimed at small and medium size enterprises or individuals. In this model, the software is made available under an open source license, and the company sells services consisting of access to an operating instance of the software. A good example is Wordpress.com, which makes web sites based on WordPress available to both free and paying subscribers.
- **Widget Frosting.** Some companies release open source software to boost sales of complementary revenue-producing

[7] In contrast, note Confluent, which is built around the open core of Kafka, which is an Apache Foundation project.

[8] https://mariadb.com/bsl11/.

products like hardware devices (such as specialized computers or consumer electronics). This model can help to increase the functional life of the devices, whose driver software may otherwise fall into obsolescence when the manufacturer retires the devices.

The New Wave of Source Available Licensing

2018 was a watershed year for open source software businesses. The year saw about a dozen significant acquisitions and IPOs, proving—if it was ever in doubt—that open source is a powerful basis for a business. But many of the most successful commercial open source companies were beginning to complain about "strip-mining"—the practice of cloud services providers of selling access to the software, keeping their improvements private, and not contributing back to the original project or doing business with the company developing it. This placed these companies in the unenviable position of serving as unpaid development and maintenance shops for some of the world's biggest and most profitable companies.

Most of the companies experiencing this pressure employed open core business models, and almost all of them were in the database sector, or a similar "middleware" software market that was of particular use to large-scale enterprise computing. These companies reacted by shifting some of their open source code to licenses that provided source code, allowed modification and redistribution, and were deployed in the frictionless fashion of open source licenses, but imposed license conditions. Because license conditions violate the Open Source Definition, these are not open source licenses, but there is no one standard term for what they are; perhaps, *source available* licenses or limited source code licenses.

In a way these licenses were nothing new, because limited source code licensing is merely a species of proprietary licensing. But perhaps because the companies implementing them were commercial open source ventures, great controversy erupted over this trend, with at least

one commentator predicting that it "will destroy open source"—a death the rumors of which were greatly exaggerated. Because there was no standard license available for this purpose, each of the companies taking this step created a custom license.[9]

A full explanation of this trend, and the licenses that resulted, are beyond the scope of a book about open source licensing. However, this trend resulted in part in the creation of a suite of licenses called the PolyForm licenses, which are source available limited licenses that are free for anyone to adopt.[10]

Good and Not Evil: The Rise of Ethos Licensing

Morality or ethos licensing is, in a way, the flip side of source available licensing. The progenitor of this class of license was the "Good not Evil" license of JSON (JavaScript Object Notation), a widely used format for storing and transporting data between servers and web pages. It said:

> Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions: The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software. The Software shall be used for Good, not Evil.[11]

This license is a conventional MIT-type license until its last line: "The Software shall be used for Good, not Evil." It is widely understood to violate the open source definition. The FSF's page "Various Licenses and Comments about Them" makes this comment about the JSON

[9] The most notable are the Redis Source Available License, the Confluent Community License, and the Elastic License.

[10] www.polyformproject.org.

[11] https://www.json.org/license.html.

license: "This is a restriction on usage and thus conflicts with freedom 0. The restriction might be unenforceable, but we cannot presume that."

FSF's comments represent the prevailing view about ethos licenses. They are open source licenses, overlaid with restrictions or conditions that are intended to change licensee behavior according to the licensor's political beliefs. Freedom Zero, the most important quality of free software licenses, is "The freedom to run the program as you wish, for any purpose." Its analog in the Open Source Definition straddles a few of the elements of the definition, such as "No Discrimination Against Fields of Endeavor."

Nevertheless, the JSON license resulted in little practical controversy, because most users could convince themselves that they were not engaged in evil activities.

But recently, there has been a spate of new ethos licenses. They have ranged from amateur licenses to sophisticated attempts to comply with the open source definition. For example:

- **Anti-996 License.** A professionally drafted license with its own GITHUB discussion page, Anti-996 was intended to address working conditions for software engineers and others by requiring licensees to adhere to local labor laws, or at a minimum, the UN "Core International Labor Standards" which include prohibitions against human trafficking. "996" refers to a working day of nine a.m. to nine p.m., six days per week. This license remains the most professional and serious ethos license out there.
- **Anti-ICE License.** A "crayon license" that was issued and quickly withdrawn, this license purported to withdraw permission to use the software to any organization that contracted with US Immigration and Customs Enforcement.
- **Hippocratic License**. This license requires the licensee to use the software to do "No Harm: The software may not be used by anyone for systems or activities that actively and knowingly endanger, harm, or otherwise threaten the physical, mental,

economic, or general well-being of other individuals or groups, in violation of the United Nations Universal Declaration of Human Rights." It clearly violates Freedom Zero.

- **Vaccine License**. This license was professionally written and addresses the need for vaccinations and an implied disdain for the anti-vax movement. It was submitted to OSI in October 2019, resulting in robust exchanges on the OSI discussion lists, but its chances for approval are probably nil. The license was submitted under the name *Filli Liberandum*, which roughly translates as "free the children." The anonymity of the submission caused as much controversy as its content, and some have posted that it was submitted to test OSI's appetite for ethos licenses.

Comparing the Anti-996 License and the ICE License illustrates some of the controversy over ethos licensing and the OSD. Of the licenses listed above, Anti-996 comes the closest to meeting the open source definition. It is a thoughtfully drafted document, and its authors include a WIKI explaining some of their drafting choices. Its development team says it is "designed to be compatible with all major open source licenses." The conditions of the license say:

> The individual or the legal entity must strictly comply with all applicable laws, regulations, rules and standards of the jurisdiction relating to labor and employment ....In case that the jurisdiction has no such laws, regulations, rules and standards or its laws, regulations, rules and standards are unenforceable, the individual or the legal entity are required to comply with Core International Labor Standards.

While this is written as a condition rather than a restriction, the license is thought by some not to be OSD compliant, perhaps because the conditions do not relate to the use of the software or the exercise of the copyright. The license was never submitted to OSI for approval, so the issue of its OSI compliance was never examined in detail. Moreover, this license is clearly not compatible with GPL, because it imposes conditions that are not in GPL, and GPL expressly prohibits imposing additional conditions.

In contrast, the Anti-ICE license prohibited use by any organization that contracted with US Immigration and Customs Enforcement and specifically banned 16 organizations, including Microsoft, Palantir, Amazon, Northeastern University, Johns Hopkins University, Dell, Xerox, LinkedIn, and UPS. The project to which the license was applied, Lerna, had not been implemented by all of these companies. The license has been withdrawn but its pull request for the project is here. This prohibition was a clear violation of Freedom Zero. The prohibition had been applied by one of the developers of the project, apparently without the consensus of other developers of the project, but with the approval of the core maintainer, who later stated:

> Despite the most noble of intentions, it is clear to me now that the impact of this change was almost 100% negative, with no appreciable progress toward the ostensible goal aside from rancorous sniping and harmful drama....I am reverting the license changes. In the future, such changes (if any) will go through a much more thorough, completely public, and fair-minded process.

The project withdrew the modified terms and kicked the developer out of the project for various conduct violations.

The philosophical debate on ethos licensing continues. Free and open source software is built on the premise that use of software should be like free speech: without moral and political restrictions. Today's developers, however, being just as politically polarized as the society at large, are attempting to use licensing to bring their point across.

While the causes behind ethos licensing may be sympathetic, it is not an effective tool to control behavior. If ethos licenses impose license conditions, then the only real remedy for violating them is an injunction not to use the software. There is no legal mechanism to curb immoral behavior, nor compel good behavior, with a license condition. In contrast, licenses that impose commercial conditions go to the traditional core of copyright, and the damages or injunctions available for violating them directly address the aims of the licensors.

That having been said, the Anti-996 License was as much an exercise in advocacy as in licensing, and in that respect, it worked, generating robust interest and discussion about working conditions in

the tech industry and beyond. As a corollary, it showed that GITHUB was not merely a development tool, but a significant vehicle for community among developers.

E-Book, Forms, and Checklists

This book is available in hard copy and in Kindle format on Amazon.com. If you would like an electronic copy of this book in pdf form, please visit my web site at www.heathermeeker.com. My web site includes a link (currently located at the "Links" tab) to sign up for an e-mailing list that announces updates of this book and related news. The email list is low traffic and can be handily unsubscribed. The welcome email will give you the credentials to download a copy of this book on the site.

That page may also contain some useful forms, which I will update from time to time, but most of my forms have now been migrated to www.blueoakcouncil.org, which provides peer-reviewed materials for practical open source education. Please check there for forms like an open source compliance policy for your company, checklists for code releases and contributions, and model open source representations. If you cannot access the materials you need, please feel free to email me at hmeeker@heathermeeker.com.

Glossary and Index

Apache Foundation is an organization that promotes open source projects, including the Apache Web Server. It is the author of the **Apache License**.

Apache License 41, 44
attribution requirements of version 1.0 107
compatibility with GPL 2 72–75
compatibility with GPL 3 169
defensive termination 206, 218, 260
patent terms 208
versions of license 54-55

API (application program interface) is the programmatic interface for software consisting of the type and name of variables that form the input or output of a routine. In object-oriented programming, it can be a class definition. 19, 22-24, 132, 138, 278

A **build script** is a set of automated instructions to a build program. A set of objects or other program elements are combined into a whole executable program via a build process. (For low-level languages, this includes linking.) 18–19, 31–36, 114, 141-143, 180

BusyBox is a set of UNIX utilities that is in all of the major Linux distros. BusyBox is licensed under GPL and has been the subject of most open source enforcement in the courts. BusyBox was originally written by Bruce Perens, but the enforcement actions were taken by later authors. 265–270, 297

A **compiler** is a program that translates source code into executable code objects. 17, 33–34, 180

CLAs (**contribution agreements** or **contribution licenses**) are inbound grants of rights to open source projects, which then usually relicense the contributed material under an open source license or under a dual license. They are usually licenses, but sometimes assignments of rights. 236–237

Copyleft describes a license with so-called viral or ShareAlike terms, sometimes called a "free software" license. It includes GPL, LGPL, MPL, and CDDL, among others. Variations include strong copyleft (usually used to describe GPL only), weak copyleft (used to describe the corporate-style licenses like MPL, Eclipse, CDDL, and often LGPL), and ultra-strong copyleft (usually used for Affero GPL) .....v, 3–4, 10-11
- basic tenets.....43
- compatibility.....65–72
- content.....59–61
- and data licensing.....315
- definition in transactions documents.....247–248
- and dual licensing.....233
- and GPL scope.....Chapter 8, 160–161
- and hardware licensing.....Chapter 21
- identification of licenses.....46–47
- and license selection.....230–232, 238–241
- and source code offers.....109–110
- and "viral".....77-78
- "weak".....42

Defensive termination is a clause in a patent license that terminates the license grant if the licensee engages in certain actions, usually including bringing patent claims against the licensor or challenging enforceable patents. In open source licensing, the trigger is usually limited to bringing a patent claim accusing the licensed software. 206–213, 217–218

Derivative works are modifications of copyrightable works of authorship, containing sufficient originality to consist of a protectable work of authorship. Derivative works are often confused with infringing works, which include not only derivative works but also works that contain unprotectable, trivial variations of the original. *Derivative work* is a term primarily used in US copyright law, and it is defined in 17 USC 101..... 11, 87, Chapter 8
and data 318
and hardware 317

Distribution (or **redistribution**) is the trigger for compliance conditions in most open source licenses. It is one of the enumerated rights of copyright in US copyright law 17 USC 106.....11, Chapter 6

Dual licensing is a business model in which the licensor offers to license a product via an open source license (usually GPL) or a proprietary license. It has mostly been supplanted by Open Core models. 58, 72, 167, 233, 239

Free software is software that meets the Free Software Definition per www.gnu.org/philosophy/free-sw.html. It is roughly the same as copyleft but sometimes only refers to GPL software. ..... ii–iv, 3–8, 51

Freedom Zero is one of the "four freedoms" of the Free Software Definition, "The freedom to run the program as you wish, for any purpose..."..... 8, 66, 328–30

FSF (Free Software Foundation) is an advocacy organization for free software. It is the steward of the GNU project and the GPL family of licenses. 25-26, 98–100, 237, 271–272, 280, 288, 297

GITHUB is the world's most popular online *versioning system and development tool, currently owned by Microsoft*. It is the

biggest collection of publicly available open source projects at this time, but also contains software under other kinds of licenses.

GPL (GNU General Public License) is the original copyleft license, stewarded by the FSF. Its latest version is version 3, but version 2 is more common. www.gnu.org/copyleft/gpl.html. ..... v, 7
and code releases ..... Chapter 16
and commercial agreements 251–253
and compatibility problems 67–75
and conditional licensing 77–78
and contracting 85–86
and copyleft 42
and derivative works ..... 11, Chapter 8
and direct licensing 44–45, 55
and distribution ..... Chapter 6
enforcement ..... Chapter 19
and exceptions 50
and notices 108-109, 112
and open source definition 7
and patents 190
version 3 ..... Chapter 10

JavaScript is a scripting language that runs code delivered by a server within a client-side browser; it should not be confused with Java, a web-oriented programming language that was written by Sun Microsystems. Java-Script is one of the often-neglected acts of distribution in SaaS systems. 18-20, 111-112, 199

LGPL (GNU Lesser General Public License) is a variation of GPL that enables use of the licensed code as a library for proprietary licenses. Its latest version is version 3, but version 2.1 is more common. www.gnu.org/copyleft/lgpl.html
compatibility with GPL ..... 68–70, Chapter 9

and copyleft....42, 51–53
and license selection....231, 239–240

Liberty or death is the nickname for one of the provisions of GPL 2, from Section 7: "If, as a consequence of a court judgment or allegation of patent infringement ... conditions are imposed on you ... that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Program at all."....49, 74, 218

A **license steward** is an organization or person who is responsible for updating an open source license and issuing new versions of it....48, 157, 281

Linking is the process by which a compiler combines object code files to make a single executable program. **Dynamic linking** is a way of linking objects that executes certain objects only as needed at run time. **Static linking** is a way of linking objects that causes all linked objects to load at program boot time and persist in memory even when the object is not being used. Linking is a means of building programs that is used primarily for low-level languages like C. 18, 29–36
and exceptions to GPL....50
and LGPL....51, 140–142, Chapter 9

Linus Torvalds is the original author of the Linux kernel. Torvalds later worked as a technology entrepreneur. He is sponsored by the Linux Foundation to work on improving Linux, and he currently holds the highest-level committer access to the project....vi, 5
and GPL 3....157, 225
and loadable kernel modules....142–145

Linux is a free software operating system, originally created to meet the specification for UNIX, particularly for microcomputer processors. It is sometimes called "GNU/Linux." .....i–vii, 3–5, 13, 23, 25–26, 34
and branding 224–225
and kernel modules 144–147
and OIN 197–198
and patent licensing 276–277

Mozilla Foundation is an organization that stewards the Firefox Browser, as well as other software promoting free web access. It also is the licensed steward for the **Mozilla Public License**. www.mozilla.org 41, 47, 71, 205, 207

Open source describes any license that fits the Open Source Definition. Open Source licenses include both permissive and copyleft licenses. The term *open source* was promoted by technologists in the 1990s to broaden the scope and appeal of open source software as compared to free software .....Preface, Chapter 1

Open Source Initiative (**OSI**) is an organization dedicated to promoting open source software. www.opensource.org..4, 7–9, 41, 61

Permissive describes licenses like BSD, MIT, Apache, and their many variations.....4, 41–42, 46–47, 51–53

Proprietary describes a license, such as an end user license, that grants restricted rights that usually do not include the right to receive, modify, or redistribute source code. This term is in ubiquitous usage, but I consider it misleading. The only true nonproprietary software, in the copyright sense, is that dedicated to the public domain.....72
and "contamination" 77

Public domain refers to material, such as a work of authorship, whose intellectual property rights have been waived by its owner. 62, 88, 138, 180, 202, 203, 225, 230

Redistribution See *Distribution*.

Richard Stallman is the inventor of free software and the author of the GNU General Public License.....v, 3–5, 93, 145, 215

SaaS (software as a service) is a business model in which a vendor sells access to software functionality, but the software runs on the vendor's computer system. Contrast this with an "on-premises" or distributed business model, in which the vendor sells licenses to copies of software to be run on the customer's computer system..... 46, 92–98, 103–104, 160, 174

SFLC (Software Freedom Law Center) is a pro bono organization that provides legal services to authors of open source software. It is involved in enforcement actions, mostly informal. 126, 267–268, 270, 297, 300

SPDX is a standard format for delivery of licensing information about software. The SPDX project is sponsored by the Linux Foundation. www.spdx.org..... 114–115

Tivoization is a nickname for a technical means of restricting access to a device or a related service if the device's software has been modified from factory condition. This was one of the concerns addressed by the "User Information" provision of GPL 3. 165–166

Tux the Penguin is a cartoon character that serves as an unofficial mascot for Linux. It graces the cover of this book. 224

UNIX is an operating system developed by AT&T Bell Laboratories for "small" systems. It was licensed under permissive licensing terms and then converted to proprietary terms, causing a forking of the system into many

incompatible "flavors." Linux was written as a free software alternative to UNIX; Linux met the interface standards for UNIX but was licensed under GPL. ii–vi, 23–26,145–146, 225, 267

Versioning of software See *Concurrent versioning system*.

Versioning of licenses means the release of successive versions of licenses by the license steward, often with the option of the licensor to select a single version, or a version and any subsequent version, released by the steward. 48–49, 156–158, 182

WTF license refers to a self-styled "very permissive" license, the "Do What the Fuck You Want to Public License." The FAQ at www.wtfpl.net/faq is entertaining reading. 58

Case Index

Alice Corp. v. CLS Bank International, 573 U.S. 208 (2014) .. 200

Anderson v. Stallone, 11 USPQ2D 1161 (C.D. Cal. 1989) 87

Artifex Software, Inc. v. Hancom, Inc., U.S. Dist. 2017 WL 1477373 (N.D. Cal. Apr. 25, 2017) 274

Apple Computer, Inc. v. Microsoft Corp., 35 F.3d 1435 (9th Cir. 1994) 135

Ashton-Tate Corp. v. Ross, 916 F.2d 516 (9th Cir. 1990) 307

Bedrock Computer Technologies, LLC v. Softlayer Technologies, Inc., et al. (filed June 16, 2009, E.D. Texas. 295

Chamberlain v. Skylink, 381 F.3d 1178 (Fed. Cir. 2004) 164

Community for Creative Non-violence (CCNV) v. Reid, 490 U.S. 730 (1989) 102

Computer Associates International v. Quest Software, Inc., 333 F.Supp.2d 688 (N.D. Ill. 2004) 266

Continuent, Inc. v. Tekelec, Inc., S.D. Cal. (July 2, 2013) 281

Data Cash Sys., Inc. v. JS&A Group, Inc., 628 F.2d 1038 (7th Cir. 1980) 91

Diamond v. Chakrabarty, 447 US 303 (1980) 189

Diamond v. Diehr, 450 U.S. 175 (1981) 192, 200

Drauglis v. Kappa Map Group 128 F. Supp.3d 46 (D.D.C. 2015)..... 294

Drew Technologies Inc. v. Society of Auto Engineers (SAE), No. 03-CV-74535-NGE-PJK (E.D. Mi. Oct. 10, 2003) 264

Edward B. Marks Music Corp. v. Jerry Vogel Music Co., 140 F.2d 267 (2d Cir. 1944)..... 306

Emerson v. Davies, 8 F.Cas. 615, 619 (D. Mass. 1845) 134

Feist Publications, Inc. v. Rural Telephone Service Co., 499 U.S. 340 (1991) 138, 318

Free Software Foundation v. Cisco Systems, Inc., No. 1:08-CV-10764 (S.D.N.Y. Dec. 11, 2008) 274

Georgia-Pacific Corp. v. United States Plywood Corp., 318 F. Supp. 1116 (S.D.N.Y. 1970) 196

Harper & Row Publs., Inc. v. Nation Enters., 471 U.S. 539 (1985)..... 90

Hellwig v. VMware, File No. 310 O 89/15 (Hamburg District Court, Germany, August 7, 2016) 282

Hewlett-Packard Co. v. Repeat-O-Type Stencil Mfg. Corp., Inc., 123 F.3d 1445 (Fed. Cir. 1997) 203

IP Innovation LLC v. Red Hat, Inc., No. 2:07-cv-447 (RRR) (E.D. Texas October 9, 2007) 295

Jacobsen v. Katzer, 535 F.3d 1373 (N.D. Cal. 2006) 199, 216, 263, 270-274, 287, 298, 306

Jin v. IChessU (Israel 2008)..... 265

Key West Hand Fabrics v. Serbin, Inc., 244 F. Supp. 287 (S.D.Fla. 1965) 308

Lexmark Int'l, Inc. v. Static Control Components, Inc., 387 F.3d 522 (6th Cir. 2004) 164

Lotus Development Corp. v. Borland International, Inc., 49 F.3d 807 (1st Cir. 1995) 137-138

MDY Industries, LLC v. Blizzard Entertainment, Inc., 629 F.3d 928 (9th Cir. 2010) 287

Metro-Goldwyn Mayer, Inc. v. 007 Safety Products, Inc., 183 F.3d 10 (1st Cir. 1999) 83

Micro Star v. FormGen, Inc., 154 F.3d 1107 (9th Cir. 1998) 137

MontaVista Software v. Lineo, Case No. 2; 02-CV-0309-J (D. Utah August 19, 2003) 286

National Car Rental Sys., Inc. v. Computer Assocs. Int'l, Inc., 991 F.2d 426 (8th Cir. 1993) 90

Oddo v. Ries, 743 F.2d 630 (9th Cir. 1984) 269, 307

Oracle America, Inc. v. Google, Inc., 750 F.3d 1339 (2014) .136, 278

Georgia-Pacific Corp. v. United States Plywood Corp., 318 F. Supp. 1116 (S.D.N.Y. 1970) 196

PPG Indus. Inv. v. Guardian Indus. Corp., 597 F.2d 1090 (6th Cir. 1979) 96

Panduit Corp. v. Stahlin Bros. Fibre Works, Inc., 575 F.2d 1152, 197 U.S.P. Q. 726 (6th Cir. 1978) 196

Pickett v. Prince, 207 F. 3d 402 (7th Cir. 2000) 86

Planetary Motion, Inc. v. Techplosion, Inc., 261 F.3d 1188 (11th Cir. 2001) 265

ProCD v. Zeidenberg, 86 F.3d 1447 (7th Cir. 1996)..... 82

Progress Software Corp. v. MySQL AB, 195 F. Supp. 2d 328 (D. Mass. 2002) 87, 264

RSO Records, Inc. v. Peri, 596 F.Supp. 849 (S.D.N.Y. 1984) .. 131

SAS Institute Inc. v. World Programming, Ltd. (2013 EWCA Civ 1482) 288

SCO Group, Inc. v. International Business Machines Corp., 2:03CV0294 DAK (D. Utah August 21, 2006) .. 198, 284, 303

SQL Solutions, Inc. v. Oracle Corp., 1991 U.S. Dist. LEXIS 21097 (N.D. Cal. 1991) 96

Sega Enterprises, Ltd. v. Accolade, Inc., 977 F.2d 1510 (9th Cir. 1992) 86, 138

Sid & Marty Krofft Television Productions, Inc. v. McDonald's Corp., 562 F.2d 1157 (9th Cir. 1977) 135-136

Silvers v. Sony Pictures Entertainment, Inc., 402 F.3d 881 (9th Cir. 2005) (en banc) 84

Software Tree, LLC v. Red Hat, Inc. 6:2009cv00097 (E.D. Texas March 3, 2009) 295

State Street Bank v. Signature Financial Group, 149 F.3d 1368 (Fed. Cir. 1998) 193

Twin Peaks Software, Inc. v. Red Hat, Inc., Civ. 4:12-cv-00911 (N.D. Cal. May 23, 2012).....275

Trubowich v. Riverbank Canning Co., 182 P.2d 182 (Cal. 1947).....96

Vault Corp. v. Quaid Software, Ltd., 847 F.2d 255 (5th Cir. 1988)..... 138

Wallace v. Free Software Foundation, Inc. (S. Dist. Ind., October 28, 2005)..... 305

Wang Lab. v. Mitsubishi Elecs. Am., 103 F.3d 1571 (Fed. Cir. 1997) 202

White v. Kimmell, 94 F. Supp. 502, 505 (S.D. Cal. 1950) 91

Worlds of Wonder, Inc. v. Vector Intercontinental, Inc., 653 F. Supp. 135 (N.D. Ohio 1986) 134

Worlds of Wonder, Inc. v. Veritel Learning Systems, Inc., 658 F. Supp. 351 (N.D. Tex. 1986) 134

About the Author

An attorney in private practice, Heather Meeker is a 25-year veteran of Silicon Valley who specializes in intellectual property matters for technology clients in a range of industries including software, communications, educational testing, computer equipment, and medical devices. She has extensive experience in open source licensing strategies and in intellectual property and technology matters related to mergers and acquisitions. Ms. Meeker is a partner at O'Melveny & Myers in Silicon Valley, and a Portfolio Partner at OSS Capital.

Ms. Meeker served as an adjunct professor of law at Hastings College of the Law and University of California Berkeley School of Law, teaching seminars on technology licensing. A member of the American Law Institute (ALI), she has served as an adviser to ALI projects, Principles of the Law of Software Contracts (2010) and Restatement of the Law, Copyright (ongoing). She has also served as counsel for the Mozilla Foundation and other open source organizations.

Ms. Meeker graduated from Yale with a BA in economics and earned her JD from the UC Berkeley School of Law, where she served as editor in chief of the Berkeley Technology Law Journal. Prior to law school, she worked as a software engineer and a professional musician.