

Understanding Open Source Licensing: Three How-To Guides

Joel West

Open Source Software Law by Rod Dixon, Artech House, 2004, ISBN 1-58053-719-7, 308 pp., US\$96.00.

Understanding Open Source and Free Software Licensing by Andrew M. St. Laurent, O'Reilly, 2004, ISBN 0-596-00581-4, 208 pp., US\$24.95.

Open Source Licensing: Software Freedom and Intellectual Property Law by Lawrence Rosen, Prentice Hall, 2005, ISBN 0-13-148787-6, 432 pp., US\$39.99.

Since their modest beginnings in the 1980s, the free software and open source movements, represented by the Free Software Foundation (FSF) and the Open Source Initiative (OSI), have generated much interest and participation.

The term *open source* generally refers to software that differs from its commercially sponsored counterpart in three ways: a collab-

orative production approach, community governance of production, and source code's ready availability to prepare derivative works. *Open Source Software Law*, *Understanding Open Source and Free Software Licensing*, and *Open Source Licensing* consider how open source licenses resemble and differ from each other and from commercial software licenses.

After researching the open source movement for five years and writing commercial software-licensing contracts for more than 15, I thought I knew a fair amount about open source licenses. These books showed how much I had to learn.

The books have differing depth and breadth of coverage of the various licenses: none purports to cover all the OSI-approved licenses (55 exist as of this writing, not counting multiple versions). They also present the material differently. In an extended essay linking open source to broader software law issues, Rod Dixon briefly reviews eight licenses; an appendix excerpts and annotates 22 additional licenses (also included on a CD-ROM). Andrew St. Laurent's reference book is almost like a *Consumer Reports* for license shoppers, with a

ONLINE REVIEWS

- **"A Good CVS Resource for Beginners"** by Lorin Hochstein
A review of *Pragmatic Version Control Using CVS* by David Thomas and Andrew Hunt.
- **"The Only Web Services Guide You'll Ever Need"** by Art Sedighi
A review of *J2EE Web Services: The Ultimate Guide* by Richard Monson-Haefel.

www.computer.org/software/bookshelf

paragraph-by-paragraph examination of 12 licenses. Both Dixon and St. Laurent include relevant non-open source licenses, such as Sun's Java license.

Lawrence Rosen brings a unique perspective as the OSI's original general counsel. He thoroughly reviews 11 approved licenses in the context of a software licensing tutorial. While St. Laurent dives into his first license on page 14, Rosen waits until page 73, after developing relevant concepts of intellectual property, software licensing, and the role of contracts.

Freedom without restrictions

Many programmers know the major division between the two open source license types—those that allow unlimited code use, represented by the BSD (Berkeley Software Distribution), and those licenses that allow code use only if modifications are shared, epitomized by the GPL (General Public License). Rosen labels these two types “academic” and “reciprocal.”

Academic licenses such as MIT, BSD, Apache, and Rosen's Academic Free License (AFL) give open source code users the most freedom. Both open source and commercial software distributions have widely incorporated code developed under such licenses (such as Berkeley's TCP/IP implementation or the Apache Web server). These licenses provide almost the same usage freedoms as public domain—the right to modify and prepare open source or proprietary derivatives under any license. Each license's key clause is a warranty disclaimer by the code's author—common to all open source licenses—and thus these are the simplest open source licenses. Both Rosen and St. Laurent need only a chapter to cover the academic licenses.

These books' value comes from addressing open source's thorny issues. Such issues include mechanisms to enforce sharing, combining licenses, and the increasing problem with software patents.

Enforced sharing

Programmers today are generally familiar with Richard Stallman's copyleft concept. As implemented in the GPL, if

you combine GPL-licensed code with your own code and redistribute or sell the new code, you must share your changes. This addresses Stallman's greatest fear—that individuals will use his code to make a successful proprietary program. Such enforced sharing is commonly called *viral* licensing. Because that term is anathema to Stallman and the FSF, Rosen uses the “reciprocal” euphemism, while St. Laurent terms it “generational limitation”; only Dixon uses the common “viral” term.

Both Dixon and Rosen weigh in on the GPL's most controversial aspect, skeptically examining Stallman's claim that dynamic linking to a GPL subroutine (such as in an operating system library) subjects the calling program to the GPL's provisions. Dixon spends four pages interpreting this issue in light of copyright law—the book's highlight. Meanwhile, Rosen draws the copyright distinction between a combined work (such as a compilation) and a derived work, arguing that a court would interpret linking as a combined work.

Because some programmers wouldn't use a GPL library under these conditions, the FSF invented the Library GPL (later called the “Lesser GPL”), which requires sharing changes to the library but not code that uses it. All three authors examine the GPL and LGPL. Rosen and St. Laurent also cover three influential licenses that correct key GPL ambiguities—the Mozilla Public License, the

Common Public License, and Rosen's Open Source License (OSL). The MPL has been the exemplar for subsequent enforced-sharing licenses, while the CPL is the basis for licensing IBM's massive Eclipse developer tools project.

When it comes to combining code, the academic licenses again give code users more freedom than enforced-sharing licenses—pretty much any license can be combined with another. The reciprocal licenses' enforced-sharing clauses make combination much trickier: Rosen provides guidelines for the most popular licenses, St. Laurent throws up his hands, and Dixon ignores the topic entirely.

Patents vs. open source

In recent years, the greatest impetus for changing open source licenses has been the software patents specter. All developers face an unknown risk from software patents because ignorance of an existing patent is no defense, and few companies do a complete patent search prior to development. But free software faces an additional problem: you can't distribute it for free if the distributor must collect and remit a royalty to the patent holder.

Beginning with the IBM Public License (CPL's precursor), open source licenses have tackled the patent issue by threatening retaliation. A firm that sues to enforce patents for an open source project loses rights to distribute that software, use its patents, or both. Licenses provide different patent retaliation approaches, including the CPL, Rosen's AFL and OSL, and the Apache license version 2.0. The next GPL version is also expected to do so. But none of these approaches have been tested in court. Additionally, some licenses have a “mutually assured destruction” clause that threatens the entire patent portfolio including unrelated patents. Although small companies have few patents at risk, larger companies might shy away from such projects: would a pharmaceutical company participate in bioinformatics open source if it put a multi-billion-dollar drug patent at risk?

Both Rosen and St. Laurent address these issues and patent license grants that each license requires. However,

These books' value comes from addressing open source's thorny issues—including mechanisms to enforce sharing, combining licenses, and software patents.

open source patent-licensing issues seem as suitable for the do-it-yourselfer as brain surgery. Programmers can read enough to become informed consumers, but given the complexity and potential risks, they should leave actual practice to a legal professional.

Picking one

Despite their different formats, the authors have many similarities. They all cover open source principles, major open source licenses, and contrasts to commercial licenses. Alas, they all buy into the open source dogma, including as-yet-unproven claims of the open source approach's universal superiority. And, taking the most expansive view of their profession's *raison d'être*, they all believe that lawyers make good business consultants and that programmers should pay an attorney for advice on a specific situation.

The two most similar books, Rosen's and St. Laurent's, offer similar depth but cover different licenses. Rosen provides more background and comparisons, while St. Laurent provides more detailed explanation for many licenses. For Apache users, Rosen is the only author who explains the agreement for contributing code, while St. Laurent's book alone analyzes the important new Apache 2.0 license. Both books are suitable for picking a license for a new project—Rosen's to be read as a tutorial, St. Laurent's as a reference book. Meanwhile, both Dixon and Rosen offer legal opinions about these licenses' future court tests.

Which book would I recommend? I learned the most from Rosen, and it's the one I'd read cover to cover for deep knowledge. As a reference for understanding specific licenses, St. Laurent's is the easiest to use, with the best organization, typography, and consistent treatment of each license. As for Dixon's book, it offers less and costs more than the other books, making it a distant third in this field.

Joel West is an associate professor at San José State and a cofounder of the Silicon Valley Open Source Research Project. Contact him at joelwest@ieee.org.

A Complete Code-Reading Source

Angela Jury

Code Reading: The Open Source Perspective by Diomidis Spinellis, Addison-Wesley, 2003, ISBN 0-201-79940-5, 528 pp., US\$54.99.

I wanted to read *Code Reading: The Open Source Perspective* as soon as I heard about it. Most of my career has involved reading, reviewing, and excavating other people's code. I learned to do this through the school of hard knocks, so the idea of an organized book that covered this area was thrilling. In the first few chapters, I found myself wishing for a study group—not because the concepts are difficult but because the exercises demand discussion and group analysis. The exercises have clear objectives and both quick, simple answers and more complex, thoughtful ones. Many exercises don't have a correct or perfect answer (much like software).

The book is clearly written with good examples, thoughtful exercises that require you to dig in, and a practice CD-ROM of open source projects (including Apache Web server 1.3, netBSD, and `hsq1` Database Engine). Diomidis Spinellis begins with basic programming elements (highlighting cautions

and common idioms) and moves through parallelism and recursion. He covers everything from code in the small to the artifacts and processes involved with large projects and how to use those artifacts to understand and navigate a large project. A complete example concludes the book with the goal of enhancing the `hsq1` database engine to natively support a new Structured Query Language date and time function. A code sample from the CD-ROM illustrates every concept, and each code sample is broken down to show what the code does, possible errors, and alternative approaches.

Code Reading: The Open Source Perspective might not offer much for experienced code readers, but even they should take a look at the chapter on advanced control flow (recursion, exceptions, nonlocal jumps, parallelism, and so forth). For the inexperienced code reader, this is a good book to learn how to objectively read code, reduce the “not how I would do it” reaction, and learn how to differentiate coding style from unwise code constructs.

The book would make a great textbook. You might not see it in an undergraduate class, but you should, because it offers so much valuable, introductory information. Many times, I've heard new college hires assigned to find and fix bugs say, “Oh, I just rewrote the code for that functionality from scratch because I couldn't understand the original code.” Reinventing the wheel was better, faster, and cheaper than fixing the code? Of course, this doesn't mean that a real problem was found and fixed. Sometimes a complete rewrite is the best choice, but it shouldn't be a knee-jerk reaction.

A code-reading introduction such as *Code Reading* would also help individuals like a colleague of mine. He dislikes reading other people's code. It gives him immediate mental dissonance because it's not how he would have done it. He usually gets hung up on how the author breaks the functionality into classes or the author's use of the C++ Standard Template Library. He must overcome this dissonance—which takes practice, experience, and under-

In the first few chapters, I found myself wishing for a study group—not because the concepts are difficult but because the exercises demand discussion and group analysis.

standing of different possibilities—before he can really see the code. *Code Reading* sets out to give you that practice and understanding.

The book also points out something obvious that I had missed. Until recently, it was difficult to find nontrivial code systems to read because they were all proprietary. Few sources were freely available from which you could learn code reading. Now, thanks to the open source community, many good, complex systems are available. The CD-ROM included with *Code Reading* provides a nice sample of projects and applications. Sourceforge.net is the largest repository on the Internet, and freshmeat.net highlights the latest updates. So, whether or not you support open source, get out there and read!

Angela Jury is an embedded software engineer with Boeing Integrated Defense Systems. Contact her at spurio23@yahoo.com.

A Technological Account of Grid Computing

Shubhashis Sengupta

Grid Computing by Joshy Joseph and Craig Fellenstein, Prentice Hall, 2004, ISBN 0-13-145660-1, 400 pp., US\$34.99.

We could define *grid computing* as a software infrastructure that allows flexible, seamless sharing of heterogeneous resources for compute- and data-intensive tasks and provides faster throughput at lower costs. A hot topic and a new computing paradigm (albeit, the concepts come from distributed, cluster, and peer-to-peer computing), both the academic and the corporate focuses are on grid computing. We see a plethora of grid computing books in the marketplace that give holistic overviews, sketch business and e-sciences utilities, and point out grid application areas. However, as a member of the practicing community

actively involved in grid computing research and development, I eagerly looked forward to a thorough technical compendium that would address questions regarding grid computing's concept, architecture, and programming models, including the Open Grid Services Architecture and the Globus toolkit. I'm pleasantly surprised to find that Joshy Joseph and Craig Fellenstein, accomplished practitioners themselves, have done a remarkable job in this regard.

A peek into the technology

Though proprietary load-balancing and workload-management software caters to the needs of the grid and high-performance computing community, only open standards can make it possible to adopt the notion of virtual organizations. In the first three chapters, the authors succinctly capture grid computing's genesis, including the major players' problems, business and technical use cases, and collective roles to promote grid open source. Notable among these players is the Global Grid Forum, of which IBM is a member. The Global Grid Forum has been working toward building open standards, protocols, and specifications at the grid's various layers, namely, the fabric, connectivity, resource, collective, and application layers. The subsequent two chapters highlight the layers and connect the grid to equally relevant and powerful service-oriented architecture and Web

Services paradigms. It's useful to know how the underlying Web Services technologies (such as Web Services Descriptive Language extensions) are exploited in forming grid services.

The following chapters delve deeper into the OGSA by describing the grid services infrastructure, the concepts of stateful services and service data elements, service lifecycle management, service handles, notification and registry, and so forth. Chapter 10 is fairly useful in exploring OGSA's basic services, including the common management, policy, and security models. The authors use Globus to explain the grid toolkit, and rightly so; Globus is the most comprehensive, widely used reference implementation for grid services (unlike frameworks such as Legion or Condor, which mainly focus on resource management) and the most difficult one, too. Globus' embedded programming model and design patterns and an illustration of a sample grid application end the book.

More than a reference manual

The book goes way beyond being just a code cookbook; it's a first attempt at presenting many new grid services technology concepts in a developer-friendly way. The examples are good in most cases. However, a few potential areas of improvement include a better treatment of other OGSA protocols, such as resource management, and a more detailed section on data access and management stacks (perhaps even a full chapter), given how important information virtualization is to grids' enterprise adoption. The grid computing standards space could also witness a state of flux with the arrival of standards for stateful Web Services, resource modeling, and notification (WSRF and WS-notification). A roadmap to grid computing's future with OGSA would be a welcome addition. The book's size and layout are fine. The annotated examples and XML code snippets weaken the readability a bit but are explanatory.

Notwithstanding these minor observations, *Grid Computing* is a notable effort at demystifying grid computing technology basics. My team members

The book goes way beyond being just a code cookbook; it's a first attempt at presenting many new grid services technology concepts in a developer-friendly way.

and I have benefited immensely from this book. I'd recommend it to grid practitioners, developers, and managers alike.

Shubhashis Sengupta is a principal researcher at SETLabs, Infosys Technologies. Contact him at shubhashis_sengupta@infosys.com.

The Philosophy of Good Programming

Stratton Penberthy

The Art of UNIX Programming by Eric S. Raymond, Addison-Wesley, 2004, ISBN 0-13-142901-9, 560 pp., US\$39.99.

The Art of UNIX Programming is a fascinating book that examines one man's philosophical views and the Unix environment's programming and design rules. Eric Raymond outlines some basic programming tenets, giving mostly universal examples from Unix. This isn't a programming book—there can't be more than 100 lines of actual code. It's more like a 30,000-foot-overview of good programming practices.

The two things I liked most about the book were its many quotes from people involved in the early days of Unix development and the way the book evaluates tools using side-by-side comparisons.

The writings of Master Foo were a great way to end the book. With references to the Tao of Programming and AI koans, the book takes a highly philosophical programming tone that I found very enlightening. It's interesting to learn about differing system philosophies, and Raymond expresses a lot of Unix's history (and general computer history) well.

Rules

Raymond starts with 17 rules of good programming. The rules actually cover everything from program design techniques to user interfaces, including system life cycles and development tech-

niques. The diversity rule takes on an almost spiritual philosophy, but ultimately, it all comes down to Rule #1: KISS—"Keep It Simple, Stupid!"

The rules all make sense, and you'll find some of them, such as parts modularity or code clarity, in every introductory programmer's book. Others, such as separation of interfaces from engines or optimization after it works, are more advanced guidelines.

The book also tries to show the various rules' trade-offs and limits, which makes it balanced. The simplicity rule is obvious but includes "use complexity when you must." The book also includes rules that should go together and that reinforce themselves, such as transparency and discoverability.

Contributors

An interesting thing about this book is its contributors list and how it uses their words as illustration. The contributors, listed in Appendix C, include Ken Thompson and Henry Spencer, along with many other Bell Labs system developers. In fact, their quotes give structure to the Philosophy of Unix. I found their insight and "war stories" very interesting.

The book also makes references to other programming and computer system material. It includes an excellent section of references, most with Web links.

This isn't a programming book—there can't be more than 100 lines of actual code. It's more like a 30,000-foot-overview of good programming practices.

Comparisons

One way that Raymond reinforces his rules' technical details is comparing various systems or tools throughout the book. This gives the reader a much better overview. In every chapter, case studies highlight a specific rule to show how it can be or has been applied or bent.

This starts a computer-operating-systems overview, reminding us that Windows still has its roots in CP/M. The book also includes an introduction to computer hardware platforms and early system software and networks, a good tool that gives readers perspective on the subject.

The protocol formats comparison was interesting. The differences and similarities of POP, SMTP, and IMAP are laid out in case studies. I gained insight into how the Internet and email work.

When describing mini-languages, Raymond gives diverse examples of how scripting and macros can work well when designed right. Of course, he examines Emacs scripting and the use of awk and Postscript. This inspired me to look into new ways I can automate my regular tasks.

The section "The Tale of Five Editors"—ed, vi, Sam, Emacs, and Wily—was also very interesting. I've used ed, vi, and Emacs, and I find the development trees and usage differences fascinating. This section ends with thoughts on software's "right size," a tough balancing point in development.

Of course, the author also compares programming languages. The C language and compiler development was a good introduction. Here, the book's Unix-centric nature excludes mentioning Visual Basic, but not having it for comparison is a small weakness. The end comparison of open source projects' lines of code by programming language was also enlightening—it almost seems like a horse race. 🐎

Stratton Penberthy is the director of Information Systems for a large labor organization in Washington, D.C., and a Web technologies instructor at the University of Phoenix Online. Contact him at spenberthy@iuoe.org.