How Is Open Source Affecting Software Development?

Diomidis Spinellis, *Athens University of Economics and Business*
Clemens Szyperski, *Microsoft Research*

The dynamism of open source software development efforts, numerous high-profile success stories, and the novel economic, business, and legal aspects of open source software adoption are justifiably creating a stir in the development community. We software practitioners increasingly face the possibility of using or basing our work on open source components, libraries, frameworks, systems, platforms, and development environments. The possibilities range from reusing a particular subsystem such as the SQLite or the HSQLDB embeddable database engines to basing our work on a complete application server such as JBoss. The number of open source projects available to developers is staggering: 30,000 projects are registered on http://freshmeat.net, 70,000 projects (of varying quality, completeness, and stability) are hosted on http://sourceforge.net, 5,400 Perl modules are on the Comprehensive Perl Archive Network (www.cpan.com), and 10,000 ports, all regularly regression tested for correct compilation and installation, are dis-

0740-7459/04/$20.00 © 2004 IEEE

tributed with the FreeBSD operating system. (Many projects appear on more than one of the locations just listed.) Following the reused open source code's evolution and deploying the corresponding components are also becoming less haphazard operations, with mechanisms such as installable packages and anonymous Concurrent Versions System (CVS) access enabling the automation of many operations. This special issue examines how the proliferation and availability of open source are affecting software development practices.

From a developer's perspective, open source is a combination of two important properties: visible source code and a right to make (relatively) unencumbered derivatives. The motivations behind the two properties are different, and each can occur in isolation—examples include Microsoft's shared source and library vendors' code licenses for developing derivative products from nonvisible source. Both properties affect—in positive and negative ways—the software artifacts (products) we develop and how we develop them (process).

Influence on software products

The most obvious boon of open source to software developers is the opportunity to base a design on existing software elements. The open source community gives us a rich base of reusable software, typically available at the cost of downloading the code from the Internet. So, in many cases we can select best-of-breed code to reuse in our system without having to reinvent the wheel. The resulting products benefit in two ways. First, the reused open source code will typically be of higher quality than the custom-developed code's first incarnation. Second, the functionality the reused element offers will often be far more complete than what the bespoke development would afford. Products can easily incorporate a standardized, sophisticated open source element such as an XML parser, a sophisticated scripting language, a regular-expression engine, or a relational database to satisfy requirements that in the past a custom-built, small-scale suboptimal implementation would have fulfilled.

Moreover, reuse granularity is not restricted by the artificial product boundaries of components distributed in binary form (which marketing considerations often impose). When reusing open source, code adoption can happen at the level of a few lines of code, a method, a class, a library, a component, a tool, or a complete system. Furthermore, when software is available in source code form, we can more easily port to our target platform and adjust its interfaces to suit our needs. Consequently, software reuse possibilities open up on three axes: *what* to reuse (promoted by the available software's breadth and price), *how* to reuse it (diverse granularity and interfacing options), and *where* to reuse it (inherent portability of source code over most binary packaged component technologies). Movement along all three axes increases the breadth of software reuse opportunities in any development effort.

In addition, source code's availability lets us perpetually improve, fix, and support the reused elements. This factor often mitigates the risk of orphaned components or incompatible evolution paths that are associated with the reuse of proprietary components. Also, by incorporating the source code of a reused element into the system being built, developers can achieve tight integration and a system that can be maintained as a whole.

On the other hand, many of the reuse options that open source opens can isolate reused code from its original authors and maintainers, leading to divergent evolution paths or a fossilized code base. Consider as an example a developer reusing a regular-expression library by directly incorporating its source code into his or her application (perhaps to port it into an embedded environment where the original library couldn't compile). The developer would have to (expensively) reintroduce any new improvements and fixes made on the original library code into the modified version. The consequences can be dramatic, as in the case of the slow or lacking propagation of security fixes. In cases where the software industry is moving toward new technologies such as 64-bit architectures or Unicode character encoding, the effort of upgrading reused source code elements might mean large-scale duplication of effort. Reuse in source code form can also result in undesired coupling between separate components when programmers hack together the adopted code into a working implementation instead of using or designing modular interfaces. This problem is exacerbated when developers fail to prune reused elements and end up dragging along unneeded dead code

The number of open source projects available to developers is staggering.

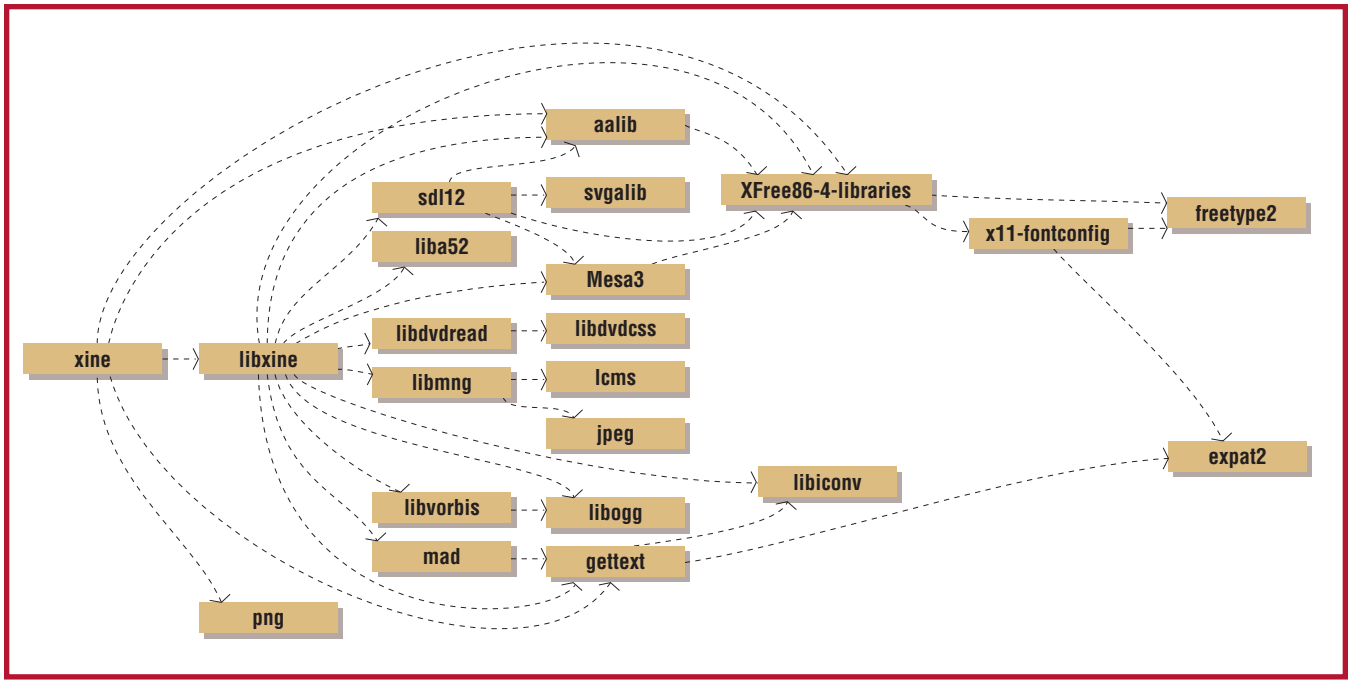

Figure 1. Library dependencies of the xine multimedia player.

that will nevertheless be compiled, distributed, and maintained with their product.

However, even when reusing open source components with precisely defined and managed interfaces (for example as a library), using that source is problematic. It's all too easy to inadvertently create deep dependencies on implementation details that will likely change in the reused element's future releases. (This problem can also occur with binary components—one prominent process that leads to such dependencies is debugging, which commonly gives away implementation details that the debugged code then depends upon.) In addition, even the published interfaces of open source components often change in ways that aren't backward compatible; rapid innovation in open source projects coupled with the lack of financial pressure from an existing customer base tend to make such changes more prevalent than on proprietary products. As an example, you can read more about how the evolution and distribution model of GNU/Linux can create problems for independent software vendors at http://primates.ximian.com/~miguel/texts/linux-developers.html.

Furthermore, the availability of source code affords an anarchic array of different dependency modalities between adopted open source elements. These include dependency requirements for specific source code, compiled libraries, packaged components, shared libraries, header files, templates, databases, plug-ins, and complete programs belonging to different open source elements. The practically zero cost of the reused elements further contributes to this commendable phenomenon of extensive reuse. As an example, Figure 1 illustrates the more than 20 library dependencies associated with the FreeBSD port of the xine multimedia player. Consider, however, restrictions that are often placed on the supported or required element versions as well as on processor and operating system platforms. Such restrictions—combined with the possibility of deploying the same reused element multiple times through different products—can easily result in incompatible dependencies that are nightmarishly difficult to track, reconcile, and maintain. By comparison, the infamous DLL problems of Windows platforms often look downright simple.

Reusing open source components can also affect the licensing model of the resultant product. Some open source licenses dictate under which license you can distribute derivative products. The Ruffin-Ebert article in this issue expands on the relevant licensing and intellectual-property issues.

An additional problem associated with reusing open source elements is that their quality varies widely from shoddy to industrial strength, and no standardized processes and metrics exist for assessing the quality of a given element. This can adversely affect products that depend on substandard software elements. Often, however, you can use the underlying source code, associated mailing list archives, and bug-tracking databases as indicators of the software's quality, degree of adoption, and support. Some software reposi-

tories, such as sourceforge.net, even provide metrics of a project's activity based on factors such as those just listed.

On the security front, products using open source can benefit from reusing widely deployed and scrutinized algorithms and protocols. However, adversaries having access to the source can more efficiently locate and exploit vulnerabilities, while the original developers may lack communication channels or resources for informing the users of their software about security vulnerabilities (see the "Keep in Touch" sidebar on page 46).

Process issues

As we indicated in the introduction, open source affects not only the products we build but also how we build them.

First of all, open source's low cost has contributed to the widespread adoption of sophisticated development platforms and tools. These include operating systems such as GNU/Linux and FreeBSD, databases such as PostgreSQL and MySQL, application servers such as JBoss, optimizing compilers such as the GNU Compiler Collection, integrated development environments such as Eclipse and KDevelop, build managers such as Make and Ant, and version control management systems such as CVS. Today, even small programming shops with a couple of developers can use sophisticated tools that once only large and well-funded development efforts could afford.

Of course, having access to sophisticated tools doesn't imply that developers actually use them. However, large open source development projects increase the visibility, accessibility, and adoption prospects of important software engineering processes such as version control, peer reviews, issue tracking, release engineering, and regression testing. These processes are standard in any CMM Level 3 and above organization, but most small shops and development efforts used to ignore them. Now, open source developers often diffuse the best-of-breed practices they learn while working in large, organized open source projects to the (possibly proprietary) projects they undertake for pay.

In addition to familiarizing themselves with sophisticated development practices, developers reusing software in source form can read the code and can often learn valuable coding practices from well-engineered software.[1] Dick Gabriel and Ron Goldman point out that ours is one of the few creative professions where writers are not allowed to read each other's work:

> *The effect of ownership imperatives has caused there to be no body of software as literature. It is as if all writers had their own private companies and only people in the Melville company could read "Moby-Dick" and only those in Hemingway's could read "The Sun Also Rises." Can you imagine developing a rich literature under these circumstances? Under such conditions, there could be neither a curriculum in literature nor a way of teaching writing. And we expect people to learn to program in this exact context?*[2]

Open-source software has changed this situation: we can now access millions of lines of code (of variable quality), which we can read, critique, and improve, and from which we can learn. In fact, many of the social processes that have contributed to the success of mathematical theorems as a scientific communication vehicle also apply to open source software.[3] Most open source programs have been

- Documented, published, and reviewed in source code form
- Discussed, internalized, generalized, and paraphrased
- Used for solving real problems, often in conjunction with other programs

On the other hand, if the reused software is not evolving, or is evolving in a direction inconsistent with the needs of the organization that uses it, the organization must contribute resources to its improvement. These resources can't always be planned in advance because the organization reusing the code typically has limited control over the reused code's development process. Disagreements with or within the software's development team often create product forks and can result in effort duplication, waste, and confusion in the community depending on the project. Forks can also occur owing to genuine irreconcilable technical considerations, in which case they might create different evolutionary paths that are all valuable to their separate user communities. As an example, at the time of this writing, software platforms based on the original BSD Unix distribution include

Many of the social processes that have contributed to the success of mathematical theorems as a scientific communication vehicle also apply to open source software.

Further Reading List

Books and articles

- *Understanding Open Source Software Development* by J. Feller and B. Fitzgerald (Addison-Wesley, 2001). Open source development from a software engineering perspective
- *The Cathedral and the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary* by E.S. Raymond (O' Reilly and Associates, 2001). The ideas behind open source
- *Component Software: Behind Object-Oriented Programming* by C. Szyperski (Addison-Wesley, 2002). The canonical reference regarding software components
- *Software Ecosystem—Understanding An Indispensable Technology and Industry* by D.G. Messerschmitt and C. Szyperski (MIT Press, 2003). Discusses software as an ecosystem of interacting participants
- "Code Reading: The Open Source Perspective" by D. Spinellis, *Effective Software Development Series* (Addison-Wesley, 2003). More than 600 examples from open source projects presenting common development and coding practices and the skill of reading code
- *Managing Open Source Projects* by J. Sandred (John Wiley & Sons, 2001). Demonstrates how the ideas and models behind the management of open source projects can apply to any type of software development
- "Release Management within Open Source Projects" by J.R. Ehrenkrantz, *Proc. 3rd Workshop Open Source Software Eng., Int'l Conf. Software Eng.* (IEEE CS Press, May 2003, http://opensource.ucc.ie/icse2003/3rd-WS-on-OSS-Engineering.pdf). Examines the release practices of three open source projects (the Linux kernel, Subversion, and the Apache HTTP server) to identify areas of weaknesses in their release management processes and suggest improvements
- "Innovation by User Communities: Learning from Open Source Software" by E. Von Hippel, *Sloan Management Rev.* (vol. 42, no. 4, Summer 2001, pp. 82–86). Presents the relationship between the open source development culture, innovation, and learning

Web sites

- **www.opensource.org.** The open source home page. It includes a list with all software licenses that classify a software distribution as open source and also a list of successful open source projects.
- **www.onlamp.com.** The LAMP Model is a widely used recipe for implementing dynamic Web sites. It entails using the GNU/Linux operating system; the Apache Web Server; the MySQL Database Engine; and Perl, Python, or PHP as a scripting language. The model demonstrates how you can use open source to implement Web-based enterprise systems.
- **http://opensource.mit.edu.** The open source directory maintained by the MIT Sloan School of Management includes a large catalog of scientific publications related to open source.
- **http://freshmeat.net, http://sourceforge.net.** These two Web sites are hubs hosting thousands of open source projects. Developers can use them to search for software to reuse, browse the corresponding forums and mailing lists, and download packages.

- FreeBSD, targeting Intel and 64-bit platforms
- NetBSD, having as its design goal portability to various hardware architectures
- OpenBSD, emphasizing security and cryptography as explicit project goals
- DragonFly BSD, experimenting with different feature sets and algorithms

Furthermore, when an organization wants or is forced to support a reused software element, it must integrate into its development process the reused element's (typically incompatible) development process. This includes issue tracking, software updates, security advisory notifications, software builds, and the actual software repository. Integrating multiple software elements (and therefore multiple incompatible software processes) can be challenging.

In addition, reusable software's widespread availability and the numerous reuse patterns afforded by open source are reducing the focus that many organizations put on centrally organized, promoted, and maintained software reuse efforts. As we discussed earlier, the ability to download open source from the Internet isn't always a proper substitute for an in-house reusable component library.

Finally, adopting open source development practices can make organizations pay less attention to strategic planning, detailed requirements elicitation, testing, and organized support. These activities are often neglected in many open source projects. Therefore organizations whose members, through their involvement in open source projects, have adopted the corresponding mindset are likely to skip or downgrade the importance of these life-cycle tasks.

Open source and non-open source development models are not at loggerheads with each other. They each have strengths and weaknesses and, most likely, they are both here to stay.

Developing with open source creates new challenges and opportunities for the products we build and the processes we use. Open source is a disruptive technology (in the sense articulated in Clayton Christensen's *Innovator's Dilemma*.[4] It currently affects development in numerous small ways, but could in the

long run lead to a paradigm shift in the way we develop software.

Acknowledgments

Vassilios Karakoidas reviewed earlier drafts of this introduction and contributed perceptive remarks. Publication of this issue's articles wouldn't have been possible without the reviewers' valuable and constructive comments. Special thanks to Terry Bollinger for shepherding the issue and to Warren Harrison, Dale Strok, and Pauline Hosillos for their support leading to a smooth editorial process. We also thank Nikos Korfiatis, who helped compile the Further Reading section, and Erast Athanasiadis, Ioanna Grinia, Alexandra-Maria Sigala, Stephanos Androutsellis-Theotokis, and Vasilis Vlachos, who performed the background research for planning this issue.

References

1. D. Spinellis, "Code Reading: The Open Source Perspective," *Effective Software Development Series*, Addison-Wesley, 2003, pp. 2–3.
2. R.P. Gabriel and R. Goldman, "Mob Software: The Erotic Life of Code," *Proc. ACM Conf. Object-Oriented Programming, Systems, Languages, and Applications*, ACM Press, Oct. 2000, www.dreamsongs.com/MobSoftware.html.
3. R. DeMillo, R. Lipton, and A. Perlis, "Social Processes and Proofs of Theorems and Programs," *Proc. 4th ACM Symp. Principles of Programming Languages*, ACM Press, 1977, pp. 206–214.
4. C.M. Christensen, *The Innovator's Dilemma: When New Technologies Cause Great Firms to Fail*, Harvard Business School Press, 1997.

About the Authors

Diomidis Spinellis is an assistant professor in the Department of Management Science and Technology at the Athens University of Economics and Business. He has written a number of open source tools and libraries, some of which are part of FreeBSD and the X Window system. He has also been a developer and manager in large commercial software projects. He is also the author of *Code Reading: The Open Source Perspective* (Addison-Wesley, 2003). Contact him at Athens Univ. of Economics and Business, Patision 76, GR-104 34 Athens, Greece; dds@aueb.gr.

Clemens Szyperski is a software architect at Microsoft and is affiliated with Microsoft Research, where he furthers the principles, technologies, and methods supporting component software. He is also an adjunct professor in the Swiss Federal Institute of Technology's School of Computing Science. He is the author of *Component Software: Beyond Object-Oriented Programming* (Addison-Wesley, 2002) and the new book *Software Ecosystem: Understanding an Indispensable Technology and Industry* (MIT Press, 2003). He received his PhD in computer science from the Swiss Federal Institute of Technology in Zurich. Contact him at Microsoft Research, One Microsoft Way, Redmond, WA 98052; cszypers@microsoft. com.