feature

open source

# Open Source Reuse in Commercial Firms

**T.R. Madanmohan and Rahul De'**, *Indian Institute of Management Bangalore*

Open source software provides organizations with new options for component-based development. As with commercial off-the-shelf software, project developers acquire open source components from a vendor (or a community) and use them "as is" or with minor modifications. Although they have access to the component's source code, developers aren't required to do anything with it. If the component's community is large and active, the adopting organization can expect frequent software updates, reasonable quality assurance, responsive bug fixes, and good technical support. Also, having freely available source code addresses two typical concerns with using COTS components: unknown implementation quality and long-term vendor support.

Despite the open source drawbacks—principally, the lack of architectural, design, and behavioral knowledge that comes with custom-built software—the use of such components in commercial software development is rising.[1] For example, network security appliances designed to do firewalling, intrusion detection, and other such functions often rely extensively on open source operating systems and utilities. As the "Example Applications" sidebar shows, many other products include open source components as well. Nonetheless, open source components raise a spectrum of issues, from requirements negotiation to product selection and product integration.

Researchers have proposed several structured, formal, or semiformal selection procedures[2] that suggest various attributes to consider when choosing an open source component. However, there's no empirical analysis of open source component use or of how commercial developers choose a component. Moreover, researchers have yet to formalize open source software use in commercial software development as an established practice.

**Using open source components raises many issues, from requirements negotiation to product selection and integration. A recent study of projects using open source revealed component selection criteria, best practices, and other related issues.**

To explore these issues from an empirical perspective, we conducted a study based on structured interviews with project developers in large and medium enterprises in the US and India. Our goal was to better understand the practices that successful commercial projects use when they incorporate open source components. Our attempt here is not only to report our survey results, but also to develop a best-practice approach to open source development based on empirical analysis. Our project sampling is limited, and doesn't represent the whole IT industry. Our goal is to empirically explore a key software industry topic, collect facts, and derive well-substantiated theses from these facts. In publishing these theses, our aim is to stimulate further investigation from both the practitioner and research communities.

## Example Applications

The following products include open source components:

- Objectweb offers several middleware solutions that embed third-party open source components.
- Virage, a leading provider of video and other rich media communication software, has embedded the MySQL database in VS Learning, the company's new product for the learning and training market.
- Apple's new Macintosh OS X is based on Free Berkeley Software Design (BSD) 3.2 and Carnegie Mellon University's Mach 3.0 project.
- Apache, Berkeley Internet Name Daemon (BIND), Sendmail, and Perl are all widely used in both commercial and noncommercial products.
- C2Net Software created its commercial Stronghold Secure Web Server with Apache and OpenSSL.
- Secure Shell is a protocol for secure shell sessions over a network; it's also a proprietary product from SSH Communication Security.

## Methodology

Our empirical research followed two parallel and complementary threads: a systematic literature investigation and an exploratory field study based on structured interviews. The projects we studied were small and of short duration. We didn't deliberately select project size as a variable; it was simply a result. Open source components were typically being used where developers were seeking quick solutions. Given this, our results might not scale to larger, more complex projects.

Our objective in interviews was to understand key issues related to using open source components in the trenches, based on the day-to-day experiences of software developers and managers. Between June and September 2003, we held structured interviews with representatives from Sun, Hewlett-Packard, IBM, HCL Technologies, Motorola India, Philips India, Sancharnet, Mahiti, Digital India, ImpulseSoft, Wipro, and Infosys. Sixteen developers, each associated with at least two of the 13 open source projects, provided insights into

- How open source projects evolve
- How developers locate open source components
- What types of open source components are considered for reuse

We conducted the interviews together, with one of us leading the interview and the other serving as scribe. The interviewer asked questions based on a questionnaire; the scribe recorded answers and asked for clarification when needed. We organized the interview questionnaire into five parts: the project, the software architecture, open source components, COTS selection criteria, and issues encountered.

To define the context and its limitations, we began each interview by introducing our motivation. We rigorously avoided suggesting any hypotheses during the interviews, and analyzed all results at a later stage to avoid biasing the interview process. After each interview, we wrote a transcript and sent it to the interviewee, asking for comments and corrections. To avoid overly taxing interviewees, our sessions lasted from 45 to 90 minutes. We also offered to give interviewees the results of our investigation. We conducted interviews face-to-face or through emails and videoconferencing.

Table 1 shows the key attributes of the projects we studied. The projects used a variety of products, which we classify according to architectural level:

- *Operating systems*: Linux, Solaris, QNX, Microsoft NT, FreeBSD
- *Middleware*: Visibroker, BEA Weblogic, OpenORB
- *Databases*: MySQL, Microsoft Access, Oracle DB, Microsoft SQL, Hypersonic SQL
- *Support software*: Hypertext Preprocessor, Apache, Tomcat, IBM Java Runtime Environment, WebMacro, Log4J, Xerces,

**Table 1**

**Project descriptions**

| Project | Description | Number of project staff | Effort (months) | Developers' location | Client country |
|---|---|---|---|---|---|
| A | Network drivers for a legacy system | 4 | 2 | India | Australia |
| B | Thin-client device | 3 | 2 | India | UK |
| C | Open source driver for a low-power battery device | 3 | 4 | US | US |
| D | Mail reflectors | 2 | 1 | India | Canada |
| E | Decoder for a media node | 2 | 1 | India | Europe |
| F | Real-time operating system switch | 3 | 2 | US | US |
| G | Integrated workflow manager | 4 | 3 | India | Germany |
| H | Extranet device | 3 | 2 | US | US |
| I | XBoss device driver | 2 | 1 | India | US |
| J | Adaptor solution for network management solution | 3 | 1 | India | US |
| K | Negotiation support system for a trading system | 4 | 3 | India | UK |
| L | Auction and recovery management system | 5 | 2 | India | US |
| M | An object processor that formats Extensible Style Sheet Language (XSL) | 4 | 2 | India | Europe |

Agent++, Intel COPS, Client, Telia BER Coder, and XMLDB

Also, most projects used at least one well-established generic architecture and technology, such as the Linux-Apache-MySQL-PHP (LAMP) architecture.

## Our basic model

On the basis of our study, we developed a simple model of the stages involved in locating and using an open source component. As Figure 1 shows, the process typically includes three steps:

- *Collection.* Developers watch for information and explore the environment for new ideas and components. They investigate key open source discussion groups and identify possible components on the basis of their firm's product roadmap. They might give preference to open source components based on, say, BSD licenses and the ability to trace an author.
- *Incubation.* Developers create and evaluate information about the software component. They also track emergent patterns in the domain and its usage. Furthermore, the developers question information that they've collected and evaluate component usability through brainstorming and what-if analysis. This process also often involves connecting two or more ideas or components.
- *Revision.* Developers critically revise existing components through social selection. This revision involves revisiting the existing component library, estimating the value of those components, and deciding whether to add to or extend them to meet new customer requirements.

Developers don't visit these stages on a linear path, and they might return to earlier phases and iterate often. Web-based libraries, for example, might be useful at every phase, and discussions with peers might occur repeatedly during idea selection and development.

## Study results

Using our basic collect-incubate-revise model, we can now examine our study find-

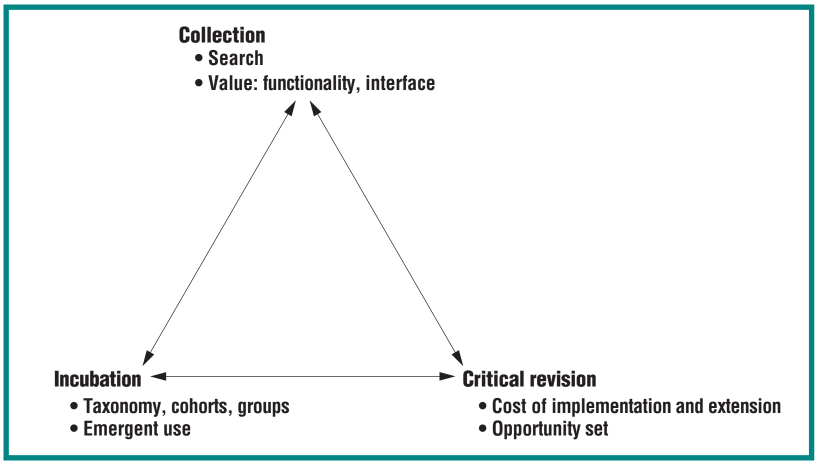

Collection
• Search
• Value: functionality, interface
Incubation
• Taxonomy, cohorts, groups
• Emergent use
Critical revision
• Cost of implementation and extension
• Opportunity set

**Figure 1. Our open source component selection model involves three basic steps: collection, incubation, and revision.**

ings on the software component selection, location, and reuse processes.

### Selecting components

Open source technologies create a fundamental conflict for software corporations: business practices dictate that software corporations retain their software's intellectual property rights, hide this IP from their competitors, and make profits on their IP investment. In contrast, the open source philosophy is to distribute source code widely and not hide the IP. Most organizations in our study therefore had to adopt a review process to study licensing issues and IP. For example, to clear an open source component or project for reuse, Motorola uses what it calls a Strategic Technology Asset Management Process, while Philips uses an Intellectual Property and Standardization review.

The developers we interviewed first screened open source components to satisfy the functionality requirements and only later addressed usability and cost issues. If the desired functionality was present, they chose the component and then worked around it to obtain required features and functionality. For example, Motorola India developers needed a thin-client device that used a touch screen and would run on a nonwindow, non-GPL platform. They sought and found open source code for the touch-screen drivers on the Internet but couldn't trace the author. The developers decided to calibrate the code and later to use it for the touch-screen driver component. Once they achieved functionality, their focus turned to quality of service. In another instance, HCL had to develop a driver for a battery-driven device. The team's developers faced issues such as the boot process's power requirements, the device's heat generation, and the boot-loader's compact size. The team found an open source module meeting these requirements and contacted the author to seek his permission for use and extension. Once the code was made available, the team went ahead improving the code to meet the project requirements.

Many of the surveyed projects that used open source or free software products didn't look at the source code, though it was easily available. The main reasons for this were that the teams had no need to look at or modify the source and didn't have enough resources (knowledge, skill, or manpower) to do so. The teams often considered the source code's availability and the existence of a wide developer community as a guarantee of assistance (if needed). Thus, familiarity was the main (and often only) attribute they considered. In several cases, developers selected both the system architecture and the open source products because they were already familiar with them. In one case (project J), for example, the choice between two network management solutions was driven by the fact that the developers knew one solution's administration interface better because they'd used the product before. When resources and time available are limited, developer familiarity seems to be the salient selection criterion for open source components.

In another case, the team selected an open source component based on its architectural compatibility. In most cases, developers first identified the key product and its core functionalities, then adopted its architecture. In other cases, they decided on the architecture before detailing the requirements. In both scenarios, once developers had chosen the architecture, it placed constraints on the products selected. Some researchers have suggested that the component selection process should be based largely on requirements and their negotiation.[3–5] While developers acknowledged that requirements are important, in these projects, the architecture was always the real selection-process driver and the fulcrum of all trade-off activities.

**The teams had no need to look at or modify the source and didn't have enough resources (knowledge, skill, or manpower) to do so.**

### Locating open source modules

Locating the right open source code isn't an easy task. It's difficult to search through all the libraries, identify and extract the best modules, and correctly estimate the customization and integration cost. We analyzed the developers' main methods for locating code as follows:

- *Formal methods* are useful only for small code, and they require formal descriptions that are costly and time consuming to produce.
- *Artificial intelligence methods* require automated use of function names, and their hit rates and precision are issues (projects that require multilingual code suffer the most).
- *Ontological approaches* are mostly in beta stages (see, for example, www.amosproject.org). Such approaches exact a high cost

for package and dictionary development and require extensive customization.

Finally, most groups we interviewed used some form of a *manual (brute force) method*. Undirected Google and Freshmeat searches were the most prevalent approaches. While this method is simple and fast, its disadvantages are that the search engines can't differentiate between package varieties. Corporate portals were another effective source, wherein a small group of evangelists promote awareness of open source and follow related developments. Digital India, for example, had a policy of storing information about source code in XML files on a corporate Linux source database. Companies often used mail reflectors and add-ons to apprise developers of corporate source topics and submissions.

**If the open source component offers the best solution and reliability for the price, then it's the most appropriate.**

### Reusing components

The type of open source component you might choose depends largely on project scope and requirements. Also, commercial organizations view open source not only in terms of cost and time but also in terms of the target component's strategic importance to the company. If the open source component offers the best solution and reliability for the price, then it's the most appropriate. For example, Philips platforms use VxWork's pSOS as the operating system, which adds to the product's cost, while using Linux in its media node's DivX decoder reduces the cost. Adapting an open source component can provide technical capabilities or evolve as a hedging strategy for an organization by balancing its technology portfolio.

Based on the discussions we had with developers, we identified five critical issues for reusable open source components: cost, customization requirements, component characteristics, licensing, and maintenance and support.

***Cost.*** In two of the study projects, the price of open source components was about 1/1,000 of proprietary software. In another case, a negotiation and recovery system was about $125,000 on a proprietary platform system that would deprecate within two years, while a similar open source system was $45,000 (using software such as Linux or FreeBSD, the Apache Web Server, the MySQL relational database, and a combination of Java, Perl, PHP, and Python). Although cost is an obvious reason for choosing open source components, many firms chose them to improve performance. Another attraction is that the marginal cost of scaling up is zero: open source software doesn't require additional licenses as an installation grows.

***Customization requirements.*** Open source components are ideal when the software solution requires high customization and using an open source component is cheaper than using a COTS product.

High customization costs and limited test-case availability might discourage developers from choosing an open source component, especially if it's a "platform product." If the requirements are robust, however, an open source component might prove to be a boon. For a real-time operating system project, for example, an open source component that

- Runs on a variety of platforms and CPUs
- Easily integrates with an integrated development environment
- Has real-time capabilities

is an ideal choice compared to a proprietary product.

***Component characteristics.*** An open source software project's modularity contributes to the approach's effectiveness, letting managers sidestep Brooks' law. However, reusability has several important criteria for success:

- Flexibility and design consistency
- Component introspection and persistence
- Measures for stability, reliability, and auditability

Open source projects that are too platform-specific aren't good either. For example, many open source content management system developers have based their spawning, multiple (often competing), and derivative projects on a single platform. To develop them into useful applications requires excessive, code-based customization. While extensibility is important, customers expect to see the inclusion of core features, together with the ability to configure key settings. Open source components with low code volatility, high platform heterogeneity, and high configuration and optimization space are the best choices. Robust test cases and user credibility are other dimensions developers must consider to identify the right components.

In some cases, open source components or packages might push customers into a highly specialized environment or a single repository. For example, Midgard requires MySQL, the Red Hat CCM Core mandates Oracle, and Bricolage only uses PostgreSQL. Scripting languages (such as Perl, PHP, Python, and Tcl) bring unique advantages while dealing with different kinds of code blocks. However, they also restrict component reuse. Zope, for example, prefers its own unique database that no one outside the Python community would ever consider using.

Specific requirements reduce the solution's portability and can impact integration and the cost of implementation. External dependencies increase the complexity of using the procedure, thereby decreasing the motivation for reusing it elsewhere. The software reuse community has long espoused the notion of interface-exposing, component-oriented software reuse. The assumption is that as long as a component's interface remains unchanged, you can change its implementation to accommodate new requirements or extensions. In fact, for such reasons, David Parnas has argued that components' source code should remain hidden from component users and only the interface should be exposed.[6] Many firms adopt information hiding[7] when they hide the implementation issues; the interface also hides the underlying assumptions. Some open source groups spell out specifications, but these tend to lag behind the implementations.

***Licensing.*** An open source component's licensing terms affect its reusability. Opensource.org (www.opensource.org) lists 21 approved licenses, which are the standard licenses that the Open Source Initiative (OSI) has certified as valid. Some licenses allow noncommercial software use for free; others (such as Gnu General Public License, or GPL) are free but limit commercial reusability. GPL lets developers charge fees for support or warranty services. Whereas GPL prevents future private use of modified GPL-licensed software, BSD-type licenses even let developers create future binary-only (no source) and commercial products from BSD-licensed software. The only restriction is that developers include the original's copyright or license notice and warrantee disclaimer in the derived products. The BSD license doesn't include a termination clause covering violations of the license terms.

The base license determines how a developer can use an open source software component. Licenses that are *closed source* restrict any reuse at all, while licenses such as GPL restrict the developer's freedom, especially if the component is modified (becomes a "derivative work") or the components are not compiled together or statically linked.[7] In our study, however, some firms adopted a strategy of developing modules containing GPL-based components with APIs exposing them to avoid GPL's viral nature. In this methodology, any component can be under GPL but can be called using APIs.

Some licenses might restrict the distribution of modified source code unless it allows the distribution of patch files with the source code to modify the program at build time. Other licenses might require derived works to carry a different name or version number from the original software. From a commercial firm's perspective, the license must explicitly permit the distribution of software built from modified source code.

**From a commercial firm's perspective, the license must explicitly permit the distribution of software built from modified source code.**

The developers we interviewed preferred the BSD, MIT, and X licenses. Code reusability is also possible with the CMU, Sun Community, and other published source licenses. Some of the developers felt third-party indemnification options reduced risks associated with open source and could influence them in adopting such technologies. For example, JBoss Group announced it will indemnify and defend JBoss customers from legal action alleging JBoss copyright or patent infringement. Other vendors of open source software—including Hewlett-Packard, Red Hat, and Novell—also offer indemnifications. Moreover, projects and components working under the BSD license that have closed-source versions can add "extras" that enhance the software's features and functionality, thus creating value for the developer and the firm.

Such opportunities open windows for exploiting code forking. The various BSD versions (BSD/OS, FreeBSD, OpenBSD, and NetBSD) attest to its many descendants, or at least to the ones retaining the BSD name. There is a financial incentive to fork here; BSD/OS has no free versions, only commercial

**Software maintenance includes adding and subtracting system functionality, debugging, restructuring, conversion, tuning, and migration.**

sales. Other studies also reiterate the BSD license's popularity.[8,9] BSD21 is very popular because it lets a firm do anything with the code.[10] The license is very short and places no restriction on redistribution and use in source and binary forms, with or without modifications. It lets firms redistribute derivative works under any license scheme. Moreover, it lets developers prepare derivative works by mixing BSD and proprietary code.[11] In accordance with the OSI's goals, this license reduces the gap between the open source community and the proprietary software world. As a consequence, it's highly suitable for commercial purposes. The projects we studied used BSD networking code in all operating systems—Unix, Windows, and OS/2. Sendmail and Apache, both of which use BSD-derived licenses, were ranked as the most popular and heavily used by all programmers. The Artistic License offers a number of alternatives to code forking, provided the forking is plainly marked.

***Maintenance and support.*** Software maintenance includes adding and subtracting system functionality, debugging, restructuring, tuning, conversion, and migration. Modifications or updates are expressed as a tentative new version that might survive redistribution and subsequently be recombined and re-expressed with other new mutations to produce a new-generation version. Because most open source software is produced not by companies but by loosely coupled groups of individuals, mailing lists are the dominant support mechanism. Also, as David Nichols and Michael Twidale write:[12]

> *Given the interests and incentives of developers, there is a strong incentive to add functionality and almost no incentive to delete functionality, especially as this can irritate the person who developed the functionality in question. Worse, given that peer esteem is a crucial incentive for participation, deletion of functionality in the interest of benefiting the end user creates a strong disincentive to future participation, perhaps considered worse than having one's code replaced by code that one's peers have deemed superior.*

Important aspects to consider here include

- Documentation thoroughness
- The quality of activity on public mailing lists
- The code's history
- Corporate stakeholders and sponsors
- Developer responsiveness to bugs
- Project activity
- Support (commercial and self-help)
- Performance
- Known bugs

For its Web-based enterprise management solution, HP had the option of selecting from Pegasus, C+, Java Web Start, and Storage Network Industry Association C (SNIA C). Ease of adaptability, IBM's improved offerings for event handling, and consistent standardization efforts convinced HP developers to opt for SNIA C.

Support is critical to the success of open source software implementation and maintenance. To quote an HP manager, "The source code isn't the most important thing when judging if the project is viable—it's the activity level of the team and community around it. Once you have a competent community, the code gets fixed." Specifically, before selecting an open source component, developers must evaluate four key factors:

- The names and number of people working in the code's development and in executing a detailed set of test cases against the code. The involvement of many reputed people makes the code less errorprone and increases reliability, regardless of whether the code authors are traceable for code explanation. Developers in our study often used developer and user mailing lists and the number of active postings as proxies to identify vibrant communities and components with high support.
- The quality of documentation, the Web site's appearance, and adherence to established quality standards. Technical support by the community for open source might not be available immediately; the community must evolve with the company as it moves into the open source space.
- The recent activity level in the current-version system (or similar) repository.
- The number of stable code implementations and component standardization efforts by high-network firms, government agencies, and standards organizations.

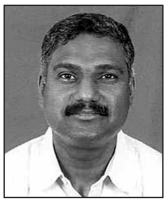
Open source products feature a variety of support mechanisms: products might have commercial support, large user and developer communities involved in developing and marketing standards, and large defense-sponsored applications. Aggregation points (for example, http://freshmeat.net) provide details about the implementation status and offer developer reviews, which are a major resource for evaluating support.

As with any software system these days, security is always an issue. However, there is little emphasis on security issues related to reusing open source components. Some of the open source systems have active developer segments that concentrate solely on security. Others seem oblivious to it. Many of the developers we talked to didn't use a specific framework or approach to assess risk and security issues.

Commercial corporations seek out open source software to boost their competitive position and gain a time-to-market advantage. However, a software manager must first consider the prime decision of whether to include open source components or not, and if so, what type. Far from offering definite answers, the projects we studied offer interesting insight into the search for an adequate product or module. This search is nontrivial and in some cases requires automated support. Our findings show that this aspect is still in its infancy, but the development of such support tools will soon be a major factor affecting open source practice in large commercial firms. Current practice appears to rely more on informal approaches and search engines. Approaches to organize, classify, and reuse the components effectively are needed.

The most important issues related to component reuse are its functionality, licensing issues, and "fit" with the existing platform. This finding isn't surprising. The questions that remain unanswered and require further study are these: What are the general parameters by which functionality and fit can be modeled? What practices can the open source community follow to effectively document and categorize their code modules to increase reuse and inclusion in other projects (assuming this is a desirable goal)? What formal and general costing model could be used to compute the cost of open source code?

## About the Authors



**T.R. Madanmohan** is an associate professor of technology and operations management at the Indian Institute of Management Bangalore and an adjunct research professor at Eric Sprott School of Business, Carleton University, Canada. His research interests include standards, open source, and service operations. Madanmohan received his PhD in management studies from the Indian Institute of Science, Bangalore. He is a member of the IEEE Engineering Management Society, Operations Research Society of India, and Society of Operations Management. Contact him at E-003, Indian Inst. of Management Bangalore, Bannerghatta Rd., Bangalore, India 560076; madan@iimb.ernet.in.

**Rahul De'** is an associate professor of quantitative methods and information systems at the Indian Institute of Management Bangalore. His research interests include multiagent modeling, IT diffusion, and e-performance metrics. De' received his PhD in artificial intelligence from the Management Laboratory of the Katz Graduate School of Business, University of Pittsburgh. Contact him at D-002, Indian Inst. of Management Bangalore, Bannerghatta Rd., Bangalore, India 560076; rahul@iimb.ernet.in.

## References

1. H. Wang and C. Wang, "Open Source Software Adoption: A Status Report," *IEEE Software*, vol. 18, no. 2, 2001, pp. 90–96.
2. P. Lawlis et al., "A Formal Process for Evaluating COTS Software Products," *Computer*, vol. 34, no. 5, 2001, pp. 58–63.
3. A.W. Brown and K.C. Wallnau, "Engineering of Component-Based Systems," Component-Based Software Engineering: Selected Papers from the Software Engineering Institute, Wiley-IEEE CS Press, 1996, pp. 7–15.
4. J. Kontio, "A Case Study in Applying a Systematic Method for COTS Selection," *Proc. 18th Int'l Conf. Software Eng.* (ICSE 96), IEEE CS Press, 1996, pp. 201–209.
5. B. Boehm and C. Abts, "COTS Integration: Plug and Pray?" *Computer*, vol. 32, no. 1, 1999, pp. 135–138.
6. D. Parnas, "On the Criteria To Be Used in Decomposing Systems into Modules," *Comm. ACM*, vol. 15, no. 12, 1972, pp. 1053–1058.
7. D.K. Rosenberg, *Open Source: The Unauthorized White Papers*, John Wiley & Sons, 2000.
8. J. Lerner and J. Tirole, "Some Simple Economics of Open Source," *J. Industrial Economics*, vol. 50, no. 2, 2002, pp. 197–234.
9. G. Robles et al., "Who Is Doing It? A Research on Libre Software Developers," *Fachgebiet für Informatik und Gesellschaft TU-Berlin*, Aug. 2001; http://widi.berlios.de/paper/study.html.
10. S.H. Lee, "Open Source Software Licensing," Apr. 1999, http://eon.law.harvard.edu/openlaw/gpl.pdf.
11. M. Fink, *The Business and Economics of Linux and Open Source,* Prentice Hall PTR, 2002.
12. D.M. Nichols and M.B. Twidale, "Usability and Open Source Software," www.cs.waikato.ac.nz/~daven/docs/oss-wp.html.