

Innovation Happens Elsewhere

Open Source as
Business Strategy

Ron Goldman & Richard P. Gabriel

A forthcoming book from Morgan Kaufmann Publishers.

Copyright © 2004 by Ron Goldman and Richard P. Gabriel
This is an excerpt from the final draft sent to the publisher.
All rights reserved.

Contents

Contents	iii
Preface	vii
Acknowledgements	xi
1. Introduction	1
Open source: a different way of doing business	1
Innovation happens elsewhere	2
Jumping in	3
Understanding open source	4
Communities	7
Who this book is intended for	10
Who else this book is intended for	10
2. Innovation happens elsewhere	13
Open source is a commons	13
Can the commons make a difference?	14
The commons and software	15
Open versus closed	16
Use of the commons: creativity & conversations	16
Innovation happens elsewhere	23
3. What is open source?	25
Open source in brief	25
Philosophical tenets of open source	28
Open source and agile methodologies	33
Common open source myths, misconceptions & questions. . . .	38

Open source and community	43
The secret of why open source works	50
Variations on open source: gated communities and internal open source.	55
Open source: why do they do it?	58
4. Why consider open source?	61
Business reasons for choosing to open source your code.	61
Creating your business model and following through with it.	80
Measuring success.	81
An example: the Innovation Happens Elsewhere strategy	81
Business reasons for using open source products.	87
5. Licenses.	91
What the license does.	92
What the license does not do.	93
More on copyright.	93
...and a quick word on patents	94
The licenses	94
Dual licensing	104
Supplementing the license—contributor agreements	105
Licenses for documentation	106
6. How to do open source development	111
The infrastructure needed for an open source project.	112
Software lifecycle.	119
Building a community.	127
Ending an open source project.	140
Joining an existing open source project	142
Open source within a company	145
7. Going with open source	149
Deciding to do open source	149
How to prepare to do open source at your company.	150
Getting approval from your company	164
Problems you can expect to encounter	173

8. How to build momentum	179
Marketing your project	180
Focus on your users and contributors	184
Community outreach.	196
Harvesting innovation.	198
Welcome the unexpected	198
9. What to avoid—known problems and failures	199
Not understanding open source	199
Don't needlessly duplicate an existing effort	201
Licensing issues	203
Design issues	204
Code issues	204
Trying to control too much.	207
Marketing issues	210
Tension between an open source project and the rest of your company	211
Community issues	212
Lack of resources	214
Recovering from mistakes.	215
10. Closing thoughts.	217
Appendix A: Resources.	223
Further reading	223
Websites of interest	224
Tools	225
Licenses.	225
Appendix B: Licences	227
Apache Software License	228
Artistic License	232
Berkeley Software Distribution (BSD).	235
FreeBSD Documentation License	236
GNU Free Documentation License (FDL).	237
GNU General Public License (GPL).	243
GNU Lesser General Public License (LGPL).	249

IBM Common Public License (CPL)	258
Microsoft Shared Source License for CLI, C# and JScript	262
Microsoft Shared Source License for Windows CE .NET	264
MIT or X License	267
Mozilla Public License (MPL).	268
Open Publication License	276
Sun Community Source License (SCSL)	279
Sun Industry Standards Source License (SISSL)	295
Sun Public Documentation License (PDL)	301
Appendix C: Contributor agreements	307
Apache Contributor Agreement	308
Free Software Foundation Copyright Assignment Form	310
Mozilla Contributor Assignment	313
OpenOffice.org Contributor Assignment	315
Project JXTA Contributor Assignment	316
Appendix D: Codename Spinnaker	319

CHAPTER 3

What Is Open Source?

THERE IS AN EXISTING LARGE OPEN-SOURCE COMMUNITY WITH ESTABLISHED PRACTICES AND CULTURE. Companies and others wishing to start or join an existing open-source project need to understand the ground rules and established practices for this larger community because the term *open source* sets expectations that need to be either met or explained. A good way to conceptualize this is to think of the open-source community as a foreign country with its own culture and every open-source project exists in that country. Although it might be possible to impose different cultural rules on that project, it is more likely to be easier to adopt existing customs. Therefore, understanding open source as it exists is crucial to success.

There are many definitions of what constitutes open source. The basic idea is very simple: By making the source code for a piece of software available to all, any programmer can modify it to better suit his or her needs and redistribute the improved version to other users. By working together, a community of both users and developers can improve the functionality and quality of the software.

Thus to be open source requires that anyone can get and modify the source code and that they can freely redistribute any derived works they create from it. The different licenses have various wrinkles on whether modifications must also be made into open source or if they can be kept proprietary.

Later in this book we also discuss cases that do not meet the full definition of open source but do involve some form of collaborative development of the source code, for example, Java.

Open Source in Brief

In practice, a typical open-source project uses a web or other internet site as the repository for the source code, documentation, discussions, design documents, bug and issue lists, and other artifacts associated with the project. A particular person or group is the copyright owner for the source, at least initially, and this

2 Chapter 3 *What Is Open Source?*

owner grants permission for individuals or other groups to make modifications to the source mediated through a version control system such as CVS (Concurrent Versions System). Periodic builds are automatically produced, and there are usually several versions of varying stability and age compiled and prebuilt for a number of platforms that anyone can download and try out.

Although the owners of the source have essentially total control of the project, they generally will enlist the advice of the community about its direction. Sometimes contributors retain copyright of their own contributions while licensing them under the same or compatible license, and other times the contributors assign copyright to the owners of the project.

Ownership versus licensing is an important distinction to grasp. Even when rights are broadly granted through a license with considerable freedom about what an outside party can do with the source, it is still owned by the copyright holders who can do whatever they wish with the source, including selling it, granting other licenses to it, or changing the direction of its development.

Often a project will be broken into modules, which can be thought of as logical or conceptual wholes, and a *module owner* is assigned to each. The module owner is in charge of inspecting proposed source changes and deciding whether to accept them or not. The module owner can also grant check-in privileges to developers who have earned his trust so that they can also modify the official source code. In many open-source projects, the overall owner and module owners are the primary contributors while others constitute a small percentage of the work. In general, debugging through use is the significant source of contributions from the larger community.

Some open-source projects, especially those whose source is owned by companies, establish a governance mechanism that ensures that the community has a recognized voice in decision making, which in turn makes the playing field more level than it otherwise might be.

A QUICK HISTORY OF OPEN SOURCE

In 1984, Richard Stallman formed the Free Software Foundation¹ and started the GNU project (GNU is an acronym for “GNU’s Not Unix”). He did so in response to seeing the collapse of the software-sharing community at the MIT AI Lab as everyone left to join companies making proprietary software. Stallman coined the term *free software* to express his philosophy that programmers should be allowed access to the source code so they could modify it to suit their needs. He developed the GNU General Public License (GPL) to assure that he would always

1. <http://www.gnu.org>

be able to see and modify the source code for any software he wrote, along with the modifications made by anyone else who might work on it. A number of important Unix tools and utilities have been developed as part of the GNU project, including the GNU GCC compiler and GNU Emacs.

In 1989 the University of California at Berkeley released its networking code and supporting utilities as Networking Release 1, the first freely redistributable Berkeley Software Distribution (BSD). This was followed in 1991 with Networking Release 2, which contained almost all the source code for BSD Unix (all but six files were included).

In 1991 Linus Torvalds started work on the Linux kernel under the GNU GPL and around 1992 combined it with the not-quite-complete GNU system to produce a complete free operating system. In 1995 Red Hat Software was formed to sell CD-ROMs plus offer support and services for the users of Red Hat Linux.

In 1995 a group of webmasters created the Apache project, based on the NCSA web server. Apache 1.0 was released later that year. According to the February 2004 Netcraft web-server survey, the Apache web server is more widely used than all other web servers combined and currently has over 67% of the web-server market.

In 1998 Netscape announced they would release the source code to their Navigator browser and 3 months later did so. Partly in response to the Netscape announcement, a group of leaders in the free software community, including Sam Ockman, John “maddog” Hall, Eric Raymond, and Larry Augustin, met to discuss how to better market the ideas of free software. They decided to use the term *open source* and, working with others, created the Open Source Definition,² which is based on the *Debian Free Software Guidelines* written by Bruce Perens for the Debian Linux distribution. They created an organization, the Open Source Initiative, to further the ideas of open source and to certify licenses as being true open source. There is a continuing tension between those who support *free software* and those who favor *open source* due to different philosophical outlooks.

More details on the history of open source are available online and in such books as *Open Sources: Voices from the Open Source Revolution* edited by Chris DiBona, Sam Ockman and Mark Stone or *Rebel Code: Inside Linux and the Open Source Revolution* by Glyn Moody.

HYBRID OPEN SOURCE

In pure open-source projects, all the workers are volunteers. There are rarely if ever formal processes that are followed or formal specifications produced. The

2. c.f., <http://www.opensource.org/docs/definition.html>

4 Chapter 3 *What Is Open Source?*

team may or may not have usability experts, documentation writers, testers, or project management of any variety. In some circles, this form of open source is called “all-volunteer open source.” Basically, whoever shows up is the team.

When companies start open-source projects, it is typical for those projects to be part of (or all of) a company initiative being done for strategic purposes and aims. Because many companies use a more traditional or conventional development process, such companies starting open-source projects are likely to want to engage the open-source community with some of their traditional tools and in-house experts. When this happens, we call it a “hybrid open-source project.”

For example, GE started its Visualization Toolkit open-source project,³ but they wanted to establish a strong cultural value around testing and quality. Based on some ideas from Extreme Programming, they instituted a distributed, automatic testing system that kicks in every night on the changes submitted to the project that day.

Another example is NetBeans where from the beginning Sun assigned human-computer interaction (HCI) experts, documentation writers, product managers, release engineers, and quality assurance people to work on it. These folks were trained in what open-source development was like and the ways that they would need to adapt, but nevertheless these other functions and some of the ways they normally operated within Sun were carried over into the project.

In large part, this book is about hybrid open source.

Philosophical Tenets of Open Source

Understanding the philosophical tenets of open source is simpler once you know a little about its history, which is about as ancient as anything in computing.

HISTORICAL ROOTS OF OPEN SOURCE

The open-source philosophy has its roots in the era of timesharing at large research universities such as MIT, Stanford, and CMU. In the 1960s and 1970s, these schools had research laboratories whose researchers and projects produced software—this included basic platforms such as operating systems and programming languages as well as higher-level, application-type software. This was necessary because either the research work itself was in the area of operating

3. This project is described in detail later in this chapter.

systems and programming languages or there were no suitable commercial or “professionally” created systems filling these roles.

The computers these labs used were almost exclusively timeshared machines, with a limited number of terminals connected to them. During the day, the machine was heavily used not only for research but also for writing papers and for administrative and clerical purposes. Yes, there was email in the 1960s. Imagine a 1 MIPs machine with about a megabyte of memory shared by 100–200 people. A lot of real work was done at night.

In a typical corporate office at that time, doors were normally locked at night and the only people wandering the halls were the janitors, who had keys to every office. Terminals were usually in offices, and if those offices were locked, the terminals were not available for use by anyone without a key. Almost certainly there were fewer terminals than people, and so the physical resources were shared as well as the computational ones. It was assumed that work took place during business hours and that the computational resources either sat idle at night or were running under the control of specific authorized individuals. After all, the computers were expensive and were required for the proper operation of the company.

Imagine if this were also true of university research labs. People could start large computations and head home, experimental operating systems might crash, experimental computations might crash the operating system—in short, things could get wedged easily. Moreover, there could be more people wishing to work than available terminals in unlocked areas and offices.

Suppose it's the 1960s and the terminal controlling an out-of-control computation is behind a locked door and the problem is sufficient to hamper other people working at 2 AM. What can you do? Suppose there is no way to kill or suspend the computation except by accessing the controlling terminal? Must everyone go home until the one person returns the next day? Time is too valuable, and there might be 50 people who have to go home because one person has locked his or her door. Or suppose the operating system has crashed in such a way that even rebooting couldn't fix it—the permanent context has been changed enough that the operating system software needs to be fixed (perhaps a new peripheral has been added, such as a robot arm). Do you wait until you can talk to the programmer in charge to make the change when you yourself are sufficiently expert to fix things?

No, you institute three changes from normal procedures:

- You do not allow offices to be locked.
- You create mechanisms for attaching a different terminal to an existing session.
- You permit anyone to work on any of the sources files.

6 Chapter 3 *What Is Open Source?*

This would be horrifying by some standards, but it worked just fine and created a culture that had very high productivity. First, rather than having a single or several system administrators (sys admins), such labs developed quite a few homegrown sys admins, perhaps a few dozen for a 200-person lab. These folks could do most anything sys admins do today, and in addition they were pretty darn good system programmers—typically for each part of the operating system or other system software there would be one or two experts who had created it and another three to six who were competent enough to maintain and improve it. People who joined the lab had all the source code available to look at how things were done and, in fact, to learn how to program well. What better way to learn about programming than by studying the working code of master programmers?

The result was that the machines were productively used 24 hours a day, 365 (or 366) days a year, with support. Operating systems, programming languages, and a lot of the software we take for granted today were developed incrementally over a few years' time with the full-time or part-time help of hundreds of programmers.

Social pressure and a kind of hierarchy of merit was used as a filter on who could work on what. Vandalism, bad behavior, and harmful mischievousness rarely, if ever, took place, such was the strength of the code of ethics that developed in such installations.

Here's a story we heard from the old days—it may not be true in its details, but its essence is accurate:

A new faculty member joined one of the “open source-like” labs and had a terminal in his office. Used to a more traditional environment, he started a job, locked the door, and went home for the night. The job ran away and locked up the system in such a way it needed to be killed from the terminal or the system would need to be rebooted. One of the system programmers had to break open the door, pulling the hinges out of the door jamb. The next morning the new faculty member found his door leaning against the wall, splinters all over the floor, a note on his seat explaining what happened, and another note from the director of the lab informing him that his unrestricted funds account would be charged to have the door and jamb repaired.

This is the culture from which the open-source movement emerged. The attitude is that people will behave well and that there is a common good being promoted. Source code is part of a shared experience that builds and supports the culture.

WARNING

Therefore, if as you read the following tenets you feel that they don't match how you work, how you feel, and how you could work, then you will not be able

to fit into the culture required to make a successful open-source project. In that case, you should not pursue open-sourcing your project: Your project will not succeed, your project will embarrass your company in the public eye, and you will have mangled your project in such a way that either it will have to be abandoned or you will have to abandon it.

Harsh, yes, but open source is not for everyone or for every project—at least not yet.

Let's look at some of the tenets of open source.

EVERYONE TOGETHER IS SMARTER THAN YOUR GROUP ALONE

When a producer puts on a play, audience members have no choice but to assume the role of critic: either you basically like the play or you don't, and you tell your friends what you think. But if the playwright were to turn to potential audience members at some point while writing the play and ask for help, these potential audience members would probably try their best to help and at some point might assume (in their own minds) a co-author role. It's harder for a co-author to be as harshly critical as a critic.

Inviting people to look at source code puts them in a frame of mind to help out. Not only will you gain a lot of testers, but these testers will have the source code and many of them will be able to locate the source of a problem in the code itself. The sum total of people running the software will encompass a wide variety of experience and expertise, and it is not unreasonable to assume that every type of bug in your code has been seen by someone in the pool at one time or another. It's like having a thousand people proofread a book for spelling and typographic errors.

You will mostly get highly informed and dedicated testers. The fact that the source code is available means that many of them will be at least able to develop their own workarounds to problems, and so they will stick with the software longer. But just as important, you will get a number of people who will want to contribute, often substantially, to the core source code.

A very common reaction to the idea of someone outside the organization contributing to the code base is that you don't want "a bunch of losers messing things up." Our experience is the opposite. First, there is no requirement in open source that every change proposed by someone must be taken back into the main source branch. So, you can select what you take, and you can modify or improve it if you wish. Second, it is your decision whether someone is trusted enough to check things directly into the source tree. You will have developed a relationship with all these people, looked at their work, and worked with them to improve their skills and use the style you want and, in most cases you will wish you could hire them. Third, there will not be that many of them anyway. For Linux, which is

8 Chapter 3 *What Is Open Source?*

the biggest open-source project, there are on the order of 100 people who are permitted to check code into the source tree without supervision. Many of these people “own” parts of the code or modules, and they supervise other contributors. Fourth, almost all of the people who have access to the source code will be able to identify and perhaps isolate problems, and this will make things easier on the developers. In fact, most people will hesitate to mess things up and will simply report problems.

The phrase “innovation happens elsewhere” is based on the observation that not all the smart people work for you. But suppose they do: Suppose that *all* the people out there are losers when it comes to your project and its software. What will you do if one of your developers becomes ill or leaves your company? Any replacement will have to come from this pool of losers. Do you really believe you are in this position? Better have excellent health benefits, fabulous compensation packages, and bodyguards for your developers.

OPEN SOURCE REQUIRES AN OPEN DEVELOPMENT PROCESS—STARTING WITH THE DESIGN

It might be tempting to think that open source is really just about making the source code available for viewing. If you drop code into a public location and continue development on a private copy inside your company, you are not doing open source. You are running a source access program or simply providing extra documentation. To develop the right level of trust, the code out there must be the real source code that your developers are working on. You might have various modules checked out while the code is being worked on, but that must be true on a module-by-module basis not the entire source base.

Moreover, your developers are simply part of the larger pool of developers. They are perhaps better focused, they might work together better, or they might have goals of importance to your company, but they are otherwise exactly in the same boat as all the outside developers.

This means also that many if not all of what would otherwise be internal development mailing lists need to be public. All design and implementation decisions that affect the public source code must be done in the open. There can be no distinction between us and them.

In exchange for this, you build incredible loyalty and an intense desire to move the code forward with a shared vision. You get immediate and totally relevant feedback from the market. If you are short of resources, perhaps they will be supplied from the outside. The highly influential book on people management *Peopleware* (by Tom DeMarco and Timothy Lister) suggests that a good way to

build a team is to have an introductory shared project, where the project itself comes to symbolize the group. Perhaps it's getting your newly assembled team together to cook an impromptu meal, but in the end it's working a shared artifact that does the trick. It's the same trick here. But it's no trick, it is part of the social nature of people.

THE BUSINESS MODEL MUST REINFORCE THE OPEN-SOURCE EFFORTS

You cannot charge people to use code that they are writing or have written. That means that you cannot simply take the code and charge money directly for it. You need to choose a business model that supports open source. The following are some classic open-source business models:

- Bundle open-source software with perhaps some other software and with support, charging for the bundle and for additional testing and quality.
- Add value in the form of additional modules or surrounding software and sell that additional software bundled with the open-source software.
- Provide a service based on the open-source software, such as a subscription service that updates customers' sites with tested and assured code.
- Sell consulting services that leverage the open-source software.
- Sell ancillary things such as books, T-shirts, and mugs.
- Sell hardware that runs the open-source software particularly well.
- Sell software that uses the open-source software as a platform.

There are two more business models that apply if you own the copyright to all of the source code and expect mostly minimal contributions from outside developers.

- Release the software as open source, but license the software to companies that wish to use it in a proprietary product.
- Sell the newest version, but release the previous version as open source.

Ubiquity, winning over hearts, thousands of eyes looking over the code, better platform security, and getting additional outside help are some of the reasons to do open source. The classic case is that you need to make a platform ubiquitous for some other business reason, and so you use open source as the conduit to ubiquity.

10 Chapter 3 *What Is Open Source?*

CREATING AND INVOLVING OUTSIDE DEVELOPERS REQUIRES INTERNAL RESOURCES

Many people think doing open source is a no-brainer: You can add developers to your team and don't need to pay them. However, you do need to attract, foster, and support those outside developers, and that does require internal resources. First and foremost, it requires that your internal developers take the time to participate on the public mailing lists, answer questions from outside developers, and promptly respond to bug fixes and code contributions. Your developers also need to document the system architecture so outside folks can find their way around in the source code. You may even need to rewrite the source code to be more modular, especially if it is currently one big monolithic mess.

Another cost is setting up and maintaining the infrastructure needed for an open-source project: a CVS code archive, a bug database, various mailing lists, and a website for the project. For large projects, these can each require a full-time person.

OPEN SOURCE IS A GIFT ECONOMY

To understand open source, it helps to make a distinction between a *commodity economy*, to which we are accustomed in a capitalist society, and a *gift economy*. In a gift economy, gifts are exchanged, forming a bond based on mutual obligation: In the simplest form of gift exchange, when one person gives a gift to another, the receiver becomes obligated to the giver, but not in a purely mercenary way—rather, the recipient becomes very much like a member of the giver's family where mutual obligations are many, varied, and long lasting. A person may give a gift with the realistic expectation that someday a gift of equal or greater use value will be received or that the recipient will pass on a further gift. In an open-source project, the gift of source code is reciprocated by suggestions, bug reports, debugging, hard work, praise, and more source code.

The commodity economy depends on scarcity. Its most famous law is that of "diminishing returns," whose working requires a fixed supply. Scarcity of material or scarcity of competitors creates high profit margins. It works through competition.

The gift economy is an economy of abundance—the gifts exchanged are inexhaustible. Gift economies are embedded within noneconomic institutions such as kinship, marriage, hospitality, artistic patronage, and ritual friendship. A healthy Western family operates on a gift economy. In an open-source project, the status and reputation of individuals depend on the quality of the gifts they contribute.

For open source to succeed in communities that include corporations, the source-code gift-based economy needs to thrive alongside the commodity economy just beyond its boundaries.

THE WORK-IN-PROGRESS EFFECT

The concept of the gift economy is related to what we call the *work-in-progress* effect. This effect is one of the primary reasons that writers' workshops in the literary and software-patterns worlds work as well as they do. The act of opening the curtain early when there is still time to affect the outcome seems to elicit a profound response from the audience, and the response is heightened when the invitation is to be, essentially, co-authors. Harsh criticism is replaced by constructive criticism. Responsibility becomes jointly held. The co-authors align in their regard for the work, although their individual opinions about what to do may differ quite a bit. The concern becomes how to make the work the best it can be rather than commenting on the work or the author.

Open source depends on this response.

OPEN SOURCE IS NOT BUSINESS AS USUAL

In summary, deciding to make a project open source means that it will be very different from your normal proprietary project. All decisions and design will need to be done in the open. Schedules will depend on people outside your company whom you cannot control. The community that emerges will take on a life of its own, possibly taking the project in directions you neither anticipated nor desire.

Your business model and processes must be different or else you won't succeed. You don't do open source as an add-on to your usual process. Deciding to do open source is like deciding to get married and start a family: It takes a commitment, and once you start down that path it will change how you live.

Open-Source and Agile Methodologies

During the last 5 years, a set of methodologies have become popular, called *agile methodologies*. An agile methodology is, in general, one that emphasizes incremental development and small design steps guided by frequent interactions with customers. The customer and developers get together and agree on the next set of features and capabilities for the software. Ideally, the work should take at most a few weeks. The developers then make the additions and the software is released to the customers, who react to it, perhaps making corrective suggestions.

Agile methodologies and open source would seem, at first glance, to be radically different: Agile methodologies are thought of as being about small, collocated teams and open source as being about large, distributed ones. A company might expect that the benefits of one are pretty different from the benefits of the other. Agile methodologies arose, largely, from the ranks of paid

12 Chapter 3 *What Is Open Source?*

consultants, whereas open source seems like a hippie phenomenon. A company might, therefore, believe there is a sharp choice to be made between them, but the choice has more to do with the conversations, the diversity of participants, and the transparency of the process to the outside world than it does with the philosophy of design and development: The two approaches share many principles and values.

Some agile methodologies have special practices that set them apart from others—for example, extreme programming uses pair programming and test-driven development. Pair programming is the practice of two people sitting at the same computer screen with one person typing and the other observing and commenting. Instead of one person sitting alone with his or her thoughts, pair programmers engage in a conversation while working, which serves as a real-time continuous design and code review. Test-driven development is the practice of defining and implementing testing code before the actual product code is implemented. The following are the agile development principles taken from the agile Manifesto website⁴—most of these principles also apply to open source, except as noted.

- *“Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.”*
Open-source does not talk about the customer, but in general, open-source projects do nightly builds and frequent named releases, mostly for the purpose of in-situ testing.
- *“Welcome changing requirements, even late in development. Agile processes harness change for the customer’s competitive advantage.”*
Open-source projects resist major changes as time goes on, but there is always the possibility of forking a project if such changes strike enough developers as worthwhile.
- *“Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter time scale.”*
Open-source delivers working code every night, usually, and an open-source motto is “release early, release often.”
- *“Business people and developers must work together daily throughout the project.”*
Open-source projects don’t have a concept of a business person with whom they work, but users who participate in the project serve the same role.
- *“Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.”*

4. <http://agilemanifesto.org/principles.html>

All open-source projects do this, almost by definition. If there is no motivation to work on a project, a developer won't. That is, open-source projects are purely voluntary, which means that motivation is guaranteed. Open-source projects use a set of agreed-on tools for version control, compilation, debugging, bug and issue tracking, and discussion.

- *“The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.”*

Open source differs most from agile methodologies here. Open-source projects value written communication over face-to-face communication. On the other hand, open-source projects can be widely distributed, and don't require collocation.

- *“Working software is the primary measure of progress.”*

This is in perfect agreement with open source.

- *“Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely.”*

Although this uses vocabulary that open-source developers would not use, the spirit of the principle is embraced by open source.

- *“Continuous attention to technical excellence and good design enhances agility.”*

Open source is predicated on technical excellence and good design.

- *“Simplicity—the art of maximizing the amount of work not done—is essential.”*

Open-source developers would agree that simplicity is essential, but open-source projects also don't have to worry quite as much about scarcity as agile projects do. There are rarely contractually committed people on open-source projects—certainly not the purely voluntary ones—so the amount of work to be done depends on the beliefs of the individual developers.

- *“The best architectures, requirements, and designs emerge from self-organizing teams.”*

Possibly open-source developers would not state things this way, but the nature of open-source projects depends on this being true.

- *“At regular intervals, the team reflects on how to become more effective, and then tunes and adjusts its behavior accordingly.”*

This is probably not done much in open-source projects, although as open-source projects mature, they tend to develop a richer set of governance mechanisms. For example, Apache started with a very simple governance structure similar to that of Linux and now there is the Apache Software Foundation with management, directors, and officers. This

14 Chapter 3 *What Is Open Source?*

represents a sort of reflection, and almost all community projects evolve their mechanisms over time.

In short, both the agile and open-source methodologies embrace a number of principles and values, which share the ideas of trying to build software suited especially to a class of users, interacting with those users during the design and implementation phases, blending design and implementation, working in groups, respecting technical excellence, doing the job with motivated people, and generally engaging in continuous (re)design.

A good example of a company-related open-source project that embraces both open-source and agile values is the Visualization ToolKit (VTK), which is partly sponsored by GE. VTK is a software system for 3D computer graphics, image processing, and visualization, and portions of it are subject to patents held by GE and a smaller company called Kitware. As its website⁵ states:

VTK supports a wide variety of visualization algorithms including scalar, vector, tensor, texture, and volumetric methods; and advanced modeling techniques such as implicit modelling, polygon reduction, mesh smoothing, cutting, contouring, and Delaunay triangulation. In addition, dozens of imaging algorithms have been directly integrated to allow the user to mix 2D imaging/3D graphics algorithms and data. The design and implementation of the library has been strongly influenced by object oriented principles. VTK has been installed and tested on nearly every Unix-based platform, PCs (Windows 98/ME/NT/2000/XP), and Mac OS X Jaguar or later.

The kit is substantial, encompassing over 600 C++ classes and around half a million lines of code. There are over 2000 people on the VTK mailing list. GE's stance regarding VTK as a commercial advantage is summed up in the following statement: "We don't sell VTK, we sell what we do with VTK."⁶ GE has a number of internal and external customers of the toolkit—it is used in a variety of projects GE is involved with. Kitware provides professional services associated with VTK.

As an open-source project, VTK is a bit unusual, and this is the result of some of its principals being involved with GE, which is the prime supporter of a design and implementation methodology called *six sigma*. Six sigma refers to a statistic that states that a manufactured artifact is 99.99966% defect-free, and it also refers to a process in which factors important to the customers' perception of quality are identified and systematically addressed during a design and implementation cycle whose steps are Define, Measure, Analyze, Improve, Control (DMAIC). Open source

5. <http://www.vtk.org/index.php>

6. <http://www.crd.com/~lorensen/ExtremeTesting/ExtremeTestingTalk.pdf>, p. 5.

involves the possibility of diverse innovations and also provides opportunities for interacting with customers in a direct way, which is appealing to an organization focused on customers, but there is also the possibility of erratic results when there is not a strong, explicit emphasis on quality that can be enforced. Therefore, open source went only part of the way to satisfying GE's goals for quality.

Moreover, the original VTK implementation team was small and dispersed within GE, and its members were admittedly not software engineers. The open-source component added to this the need to find a way to handle quality. The solution was to adopt some of the practices of Extreme Programming, which is one of the agile methodologies. Extreme Programming (or XP) emphasizes testing and advocates a practice called test-driven design in which tests are written at the same time as or before the code is designed and written.⁷ Writing tests first has the effect of providing a sort of formal specification—the test code—as well as a set of tests to be used for regression and integration testing. XP calls for frequent (tested) releases, and VTK combines this with the open-source practice of “release early, release often” to do nightly, fully tested builds.

The VTK developers implemented a regimen in which submitted code is tested overnight using a large corpus of regression tests, image regression tests (comparing program output to a gold standard), statistical performance comparisons, style checks, compilation, error log analyses, and memory leak and bounds-check analyses; the software's documentation is automatically produced; and the result is a quality dashboard that is displayed every day on the website. The dashboard is similar to those produced by the Mozilla project⁸, but considerably more detailed. The tests are run on around 50 different builds on a variety of platforms across the Internet, and distributions are made for all the platforms.

The reasons for this approach, as stated by the original team, are as follows:

- To shorten the software engineering life cycle of design/implement/test to a granularity of 1 day.
- To make software that always works.
- To find and fix defects in hours not weeks by bringing quality assurance inside the development cycle and by breaking the cycle of letting users find bugs.
- To automate everything.

7. There is considerably more to Extreme Programming. Kent Beck's book, *Extreme Programming Explained: Embrace Change* is a good place to learn about it, as is the website <http://www.extremeprogramming.org>.

8. <http://tinderbox.mozilla.org/showbuilds.cgi>

16 Chapter 3 *What Is Open Source?*

- To make all developers responsible for testing (developers are expected to fix their bugs immediately).

Among the values expressed by the original development team are the following:

- Test early and often; this is critical to high-quality software.
- Retain measurements to assess progress and measure productivity.
- Present results in concise informative ways.
- Know and show the status of the system at any time.

This is not all. The VTK website provides excellent documentation and a coding style guide with examples. Most of the details of the mechanics of the code are spelled out in detail. Moreover, there are several textbooks available on VTK.

In short, the VTK open-source project has integrated open-source and extreme-programming practices to satisfy GE's need to express to customers its commitment to quality, even in projects only partially controlled by GE. Furthermore, GE has tapped into a larger development community to assist its own small team, so that its customers get the benefits of a high-functionality, high-quality system infused with GE values.

CONTINUOUS (RE)DESIGN

The primary source of similarities between open source and the agile methodologies is their shared emphasis on continuous (re)design. Continuous design is the idea that design and building are intertwined and that changes to a design should be made as more is learned about the true requirements for the software. This is why both camps agree with the mantra, "release early, release often."

Continuous design is an approach that is predicated on recognizing that it is rarely possible to design perfectly upfront. The realization is that design is often the result of slowly dawning insights rather than of knowing everything at the start of the project and that, like most projects, the activities are progressive and uncertain. Specifications of software function, usability, and structure, for example, cannot be fully known before software is designed and implemented. In continuous design, software source code, bug databases, and archived online discussions capture and track the preferences and realities of co-emerging software systems and their user/developer communities in a continuous cycle of innovation, change, and design. Explicit and formal specifications and formal design processes rarely exist: The code itself along with the archived discussions are the specification.

Some open-source projects, especially hybrid company/volunteer projects, use more formal processes and produce more formal artifacts such as specifications, but even these projects accept the idea that the design should change as the requirements are better understood. In fact, we could argue that even software produced using the current principles of software design, software engineering, and software evolution are often discretized versions of continuous design—imposing the idea of formal design and specifications done largely upfront, but (unconsciously) allowing the effect of continuous design over a series of infrequent major releases rather than through small, essentially daily ones.

Common Open-Source Myths, Misconceptions, and Questions

The picture of open-source software painted by the popular media tends to be superficial and simplistic. Open source is touted as a miraculous way to produce software at no cost. For anyone developing software professionally, all this open-source hype no doubt seems pretty farfetched. Let's take a closer look and try to shed some light on what open source is really all about.

MYTH 1: OPEN-SOURCE DEVELOPMENT IS SOMETHING NEW

If you read the newspaper, open source seems to have started with the Linux operating system back in 1991 (or more likely, in 1997 or 1998 when whoever wrote the article finally heard about Linux). The actual facts are a bit different: Open source is as old as computer programming. If you had wandered into places such as MIT or Stanford in the 1960s, you would have seen that sharing software source code was assumed. Early development of the ARPAnet was helped by freely available source code, a practice that has continued as it grew into today's Internet. The Berkeley version of Unix dates from the mid-1970s. All in all quite a distinguished lineage.

The creation of software by a loosely coupled group of volunteers seems a thoroughly contemporary phenomenon, based on the free outlook of the 1960s—a kind of fallout of free love and hippiedom—but the concept of distributed group development is hardly new.

On Guy Fawkes Day, 1857, Richard Chenevix Trench, addressing the Philological Society, proposed the production of a new, complete English dictionary based on finding the earliest occurrences of each of the English words ever in printed use. That is, the dictionary would be constructed by reading every book ever written and noting down exactly where in each book a significant use of every word occurred; these citations would be used to write definitions and short

18 Chapter 3 *What Is Open Source?*

histories of the words' uses. In order to do this, Trench proposed enlisting the volunteer assistance of individuals throughout English-speaking countries by advertising for their assistance.

Over a period of 70 years, many hundreds of people sent in over 6 million slips with words and their interesting occurrences in thousands of books. This resulted in the *Oxford English Dictionary*, the ultimate authority on English, with 300,000 words, about 2.5 million citations, and 8.3 citations per entry in 20 volumes.

Compare this with the best effort by an individual—Samuel Johnson, over a 9-year period, using the same methodology and a handful of assistants called *amanuenses*, produced a two-volume dictionary with about 40,000 words and in most cases one citation per entry. As we look at these two works, Johnson's dictionary is a monument to individual effort and a work of art, revealing as much about Johnson as about the language he perceived around him, while the OED is the standard benchmark for dictionaries, the final arbiter of meaning and etymology.

MYTH 2: OPEN-SOURCE DEVELOPMENT IS DONE BY HOBBYISTS AND STUDENTS

The stereotype for the sort of person who contributes to an open-source project is that of a hobbyist or student, someone you perhaps wouldn't take too seriously. After all, full-time professional programmers don't have time for such things. Well, first, students and hobbyists can often write very good code. Next, lots of professionals do like to program on their own time. A study done by the Boston Consulting Group⁹ found that over 45% of those participating in open-source projects were experienced, professional programmers, with another 20% being sys admins, IT managers, or academics. The same study found that over 30% of these professionals were paid by their day job to develop open-source software. Both Sun and IBM have engineering teams contributing to various parts of the Apache web server. Most companies building PC peripherals now write the needed device drivers for Linux as well as Windows. In fact, the open-source process encourages the replacement of contributions from less capable programmers with code from more capable ones.

MYTH 3: OPEN SOURCE SOFTWARE IS LOW QUALITY

How can a bunch of random programmers, with no quality assurance (QA) folks, produce code with any degree of quality? Isn't open-source software full of

9. The Boston Consulting Group Hacker Survey, Release 0.73, 2002.

bugs? Well, there may initially be as many bugs in open source as in proprietary code, but because it's open more developers will actually look at the code, catching many bugs in the process. Also everyone using the code is essentially doing QA; they report on any bugs that they find, and because they have access to the source code, they often also fix the bugs themselves.

In 2003, Reasoning, Inc., performed a defect analysis¹⁰ of the Apache web server and Tomcat, which is a mechanism for extended Apache with Java servlets, by using their defect discovery tool. For Apache, the tool found 31 defects in 58,944 source lines, a defect density of 0.53 defects per thousand lines of source code (KSLC). In a sampling of 200 projects totaling 35 million lines of code, 33% had a defect density below 0.36 defects/KSLC, 33% had a defect density between 0.36 and 0.71 defects/KSLC, and the remaining 33% had a defect density above 0.71 defects/KSLC. This puts the Apache squarely in the middle of the studied quality distribution. For Tomcat, the tool found 17 software defects in 70,988 lines of Tomcat source code. The defect density of the Tomcat code inspected was 0.24 defects/KSLC. This puts Tomcat in the upper half of quality.

If you still don't believe that open-source software is of similar quality to most commercial software, just take a look at some open-source software you use every day. Assuming you make any use of the Internet, you are relying on open-source code such as BIND, which is at the heart of the Domain Name Service (DNS); or sendmail, which probably transports most email; and Apache, which as of February 2004 was the software running on over 67% of the world's web servers. Then there's Linux, which has won several awards for quality and has a notably longer mean time between reboots than some other major PC operating systems.

MYTH 4: LARGE-SCALE DEVELOPMENT ISN'T EFFICIENT

Having thousands of people fixing bugs might work, but how can you possibly coordinate the work of that number of developers? Without central control how can it possibly be an efficient process? Well, that's correct, but why does it need to be efficient? When you have limited resources, efficiency is important, but in an open-source effort with lots of developers, if some go off and write a module that eventually is rejected, it doesn't matter. Open-source efforts often progress in small steps. If several people are working on different solutions, to a problem, as long as one eventually produces a solution, you are making forward progress. If two solutions are produced, that's even better: just pick the best one. Also, with the ease of email and news groups, the various people working on the problem

10. <http://www.reasoning.com/library.html>

20 Chapter 3 *What Is Open Source?*

will probably find each other and spontaneously self-organize to work together to produce a result that is better than any of them alone could have produced—all without any central control.

MYTH 5: IF THE SOFTWARE IS FREE, THEN WHO'S GOING TO PAY YOU TO WRITE IT?

Why should your company pay you to write free software? Well, your company may already be doing that. Are you working on a product that is sold or distributed for free? Are you working on something only used internally? Is the income generated from selling the software you write greater than the cost to produce it? The profit may come from other activities. Likewise for free software. Your company will continue to make its money from selling hardware (e.g., servers, storage, and workstations); proprietary software; books; and consulting, training, and support.

For example, O'Reilly and Associates sells enough Perl books to pay the main Perl developers to work on new Perl features. Several of the main Linux developers are employed by Red Hat, which makes its money by packaging up free software. Cygnus (now part of Red Hat) sells support for the GNU compiler and debugger, which its employees continue to develop and give away. Sun sells servers, but gives away Java.

Look at the sections “The Business Model Must Reinforce the Open-Source Efforts” and (in Chapter 4) “Business Reasons for Choosing to Open-Source Your Code” for more details about how your company can make money from open-source software development. Keep in mind, however, that roughly 90% of the money spent on software development is for custom software that is never sold; commercial software represents less than 10% of the total investment.

MYTH 6: BY MAKING YOUR SOFTWARE OPEN SOURCE YOU'LL GET THOUSANDS OF DEVELOPERS WORKING ON IT FOR NO COST

That would be nice, but in reality most open-source projects have only a few dozen core developers doing most of the work, with maybe a few hundred other developers contributing occasional bug reports, bug fixes, and possible enhancements. Then there are the thousands of users, who may contribute bug reports and requests for new features. The users also post messages asking how to use the software and, in a healthy project, the more experienced users post answers to those questions. Some users may even help write documentation.

Hewlett-Packard and Intel report a 5:1 or 6:1 ratio of community to corporate developers for open-source projects the two companies have been involved with.¹¹ Our belief is that this is a little high, but it isn't too far off.

Another source of data is SourceForge, which has about 70,000 projects with 90,000 developers. The distribution of the number of developers to projects there follows a power law with almost 60,000 projects with between zero and one active developers, 3000 with three, five with 30, and one with 100. To factor out the large number of inactive or dead projects on SourceForge, a study in May 2002 by Krishnamurthy¹² looked at participation only in mature, active projects and found the average number of developers per project to be four. Only 19 out of the 100 projects studied had more than 10 developers, whereas 22% of the projects had only one developer associated with them.

It's true that you don't need to pay any outside developers who choose to work on your project. However you do need to pay the cost of maintaining the infrastructure necessary for open-source development (e.g., a CVS code server, a bug database, project mailing lists, and project website), along with the people to integrate the contributions you get. You won't get something for nothing, but for a successful open-source project you can get back more than what you put in.

MYTH 7: OPEN SOURCE DOESN'T SCALE

Experience with conventional, proprietary software development teaches that the larger the project, the greater the number of resources needed for coordination and design. For an open-source project where all the discussion is via mailing lists and where there is no formal management structure, it seems that it would be impossible to efficiently organize the developers. Hence, open source might work for small projects, but not for large ones.

In his essay, *The Mythical Man-Month*, Frederick P. Brooks states that adding more developers to a late project will just make it later. In an open-source project, developers can be added at any time with no forewarning. One issue with Brooks' Law and the various studies that have subsequently either supported or qualified it is that there is a tacit assumption about the development process. Although rarely stated, the assumption is that the development team will be made up of individual contributors, each working on a separate part of the software, forming an efficient allocation of developers to the code. As it turns out, neither extreme programming nor open source obeys that assumption. Moreover, these studies assume that developers are a scarce resource, which is not true for open source.

11. *The Open Source Software Challenge in 2001*, Stanford University Graduate School of Business Case SM 85.

12. http://www.firstmonday.dk/issues/issue7_6/krishnamurthy

22 Chapter 3 *What Is Open Source?*

Although it has been difficult to set up proper experiments to test how extreme programming affects Brooks' Law, one preliminary study (Laurie Williams, North Carolina State University, private communication) showed that when a programmer was added to create a pair-programming situation, the added programmer could immediately contribute by observing and pointing out simple errors and by asking probing questions that served to clarify the thought processes of the primary programmer. Thus, the new programmer could be productive immediately, although not as productive as a full-speed developer. The difficulty in experimental methodology is to obtain a valid comparison between an extreme programming project and a traditional one.

In an open-source project, developers are no longer treated as a scarce resource that must be used efficiently. Therefore, a developer added to a project doesn't need to have a separate part carved out. Moreover, a new developer can probably contribute immediately in the same way as in extreme programming by finding (and fixing) simple errors and asking probing questions. In his essay, Brooks points out that new developers must be trained, that larger teams require greater overhead to communicate with each other, and that not every task may be partitioned.

For an open-source project, it is important to distinguish between those developers who make up the *core team*—the module owners and few developers with check-in privileges—and the much larger number of occasional contributors. The core team is always going to be too small and all the lessons of conventional software development apply to them, including Brooks' Law. However it is with the larger group of contributors that open source changes the rules: These are the folks who can track down obscure bugs and create fixes for them, help each other to get up to speed on the code, implement features from the project's wishlist, or explore and experiment with more radical modifications—all activities that free up the core team to focus on its own main work.

Instead of controlling and scheduling developers, open source relies on the developers' self-organizing based on their interests and abilities. Instead of a management structure to coordinate everyone's work, open-source development requires resources to evaluate and integrate developer contributions. Moreover, those resources can draw on the total community of developers and are not limited to any core group. To see this, look at the success of some of the large open-source projects, such as Apache or Linux.

MYTH 8: A COMPANY DOING OPEN SOURCE GIVES UP OWNERSHIP OF ITS SOURCE CODE AND PATENTS

Your company still owns any source code that it releases under an open-source license because your company still owns the copyright. The open-source license

grants others the right to examine and use the source code, but it does not affect your company's ownership of the code. As the copyright owner, your company can release the source code under another license or use it in a proprietary product. Only if the source code were distributed containing an explicit disclaimer of copyright protection by your company would the software pass to the public domain and thereby no longer be owned by your company.¹³

However, source code contributed back to your company by outside developers is owned by the author, who holds the copyright for it. Under some licenses, such as the Sun Community Source License (SCSL), your company would be able to use the contributed code without restrictions. Under an open source license, such as GPL or the Mozilla Public License (MPL), your company would be bound by the terms of the license just like any other developer.

Similarly, your company still owns the patents embodied in any source released under an open-source license, but if your company does not explicitly talk about the uses to which any such patents may be put, others might be free to use those patents.

MYTH 9: AN OPEN-SOURCE COMMUNITY IS AN OPEN COMMUNITY

An open-source community is a community surrounding an open-source artifact, but it may not be an open community, meaning that it might not be open to anyone at all joining, and that once in the community a member might not know how to move ahead and become a leader. The community can be as closed, idiosyncratic, and undemocratic as it wants to be. The license guarantees that everyone in the community has certain rights with respect to the code, but in general it does not say anything about the community.

Open Source and Community

A successful open-source project is a community effort by people with different needs and skills and includes users, designers, programmers, testers, and others. The word *community* encompasses many meanings and is used by different people to mean many different things. For understanding open source, we find the following definition from *Community Building on the Web* by

13. Code of Federal Regulations, Title 37, Volume 1 (Patents, Trademarks, and Copyrights) Part 201, Section 201.26, Paragraph i.

24 Chapter 3 *What Is Open Source?*

Amy Jo Kim most useful:

A community is a group of people with a shared interest, purpose, or goal, who get to know each other better over time. (p. 28)

Both aspects are equally important: Conversations around their shared interest in the open source project cause the participants to learn about each other. A scattered collection of people just using a piece of software is not a community, but can become one by talking with each other to understand how to best use the software. We particularly want to distinguish a true community from a *user group*, where people come mainly to learn about a product or technology but do not interact much with each other—typically the users listen to and ask questions of one or more experts.

If you ask people connected with open source about their community and how it works, many will draw something like Figure 3.1. They will tell you about how people start as just users and how some will become more involved by reporting bugs. Some may become developers who fix bugs or make minor enhancements. A few developers then get more involved and join the ranks of the core developers, being granted the right to check-in changes to the actual project's source code. This is a very code-centric view of the open-source process as a hierarchy that has users at the periphery, occasional developers closer in, core developers even closer in, and the code at the center.

And, in fact, this view makes the hierarchy into a funnel in which the goal is to convert people from one ring in the diagram to people in the next one in—almost as if the goal were to turn people into code in the end, the highest form of existence.

A direct result of this perspective is that the actual users of the program are marginalized. Although the success of the project is often measured by the number of people that use the computer program being developed, only those people who are willing and able to talk about the code can participate in discussions on

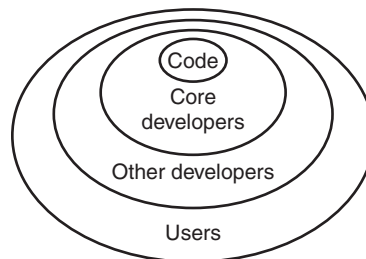


FIGURE 3.1 Single community built around source code.

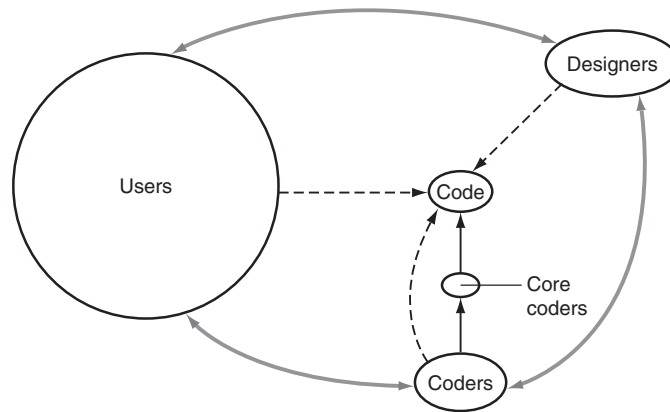


FIGURE 3.2 Different roles, still focused on code.

the project's mailing lists. Major decisions are made by those writing the code, not by those who will only use it.

If we step back a bit, we can see that the people involved in an open-source project take on various roles. Over time they may act as users, designers, or coders. So a better diagram looks like Figure 3.2. Here the code is still at the center; users, designers, and coders can look at the code, but only the *core coders* deal directly with the code. We use the term *coder* rather than *developer* to emphasize the roles being played: A coder is someone who manipulates code, either by writing new code to implement some design feature or by inspecting and correcting existing code someone else has written.

The thin solid black lines in Figure 3.2 indicate changes to the source code, the dotted lines indicate viewing the source code and interacting with the program, and the thicker, gray lines indicate communications between people in the different roles. Note that the ovals representing the different roles are not drawn to scale; the users circle should be much, much bigger for any healthy open-source project.

This is still a view that focuses on the source code, but it brings out that design is separate from coding and that designers need not be coders. It also reflects the fact that people in the community may not simply advance from user to core developer in a linear progression but adopt the roles that make sense for what they are doing at the moment.

If we step back still further and look at all the ways that people interact in the context of an open-source project, we see they do so in many different ways, such as:

- Reading, writing, modifying and debugging the actual source code.
- Reporting and commenting on bugs.

26 Chapter 3 *What Is Open Source?*

- Reading and writing documents such as user manuals, tutorials, how-tos, system documentation and project roadmaps.
- Participating in online public discussions via mailing lists, newsgroups, IRC chat sessions, and so forth.
- Visiting the project's official website or other related websites.
- Attending meetings and conferences such as user groups, community meetings, working groups, and trade shows.

Many of these do not involve the source code at all: Users can discuss how best to do their jobs, perhaps only just touching on how the project's tools can help them. Other conversations may focus on standards for protocols or application programming interfaces (APIs) that the project uses. Still other conversations may address issues affecting the larger open-source community. Each different conversation involves a different group of people. These groups may range in size from small working groups of fewer than 20 members to full communities of hundreds or thousands of participating individuals. We might represent these multiple communities as in Figure 3.3. Some people will participate in different discussions, making them members of several communities. This overlap helps to create the larger community associated with the open-source project as a whole.

Each of these communities has its own special interests; for example, some communities connected to Linux might include system administrators concerned with installing and configuring the Linux operating system on various types of computers. Another group of people might be concerned with business productivity tools (such as word processors, spreadsheets, and presentation tools)

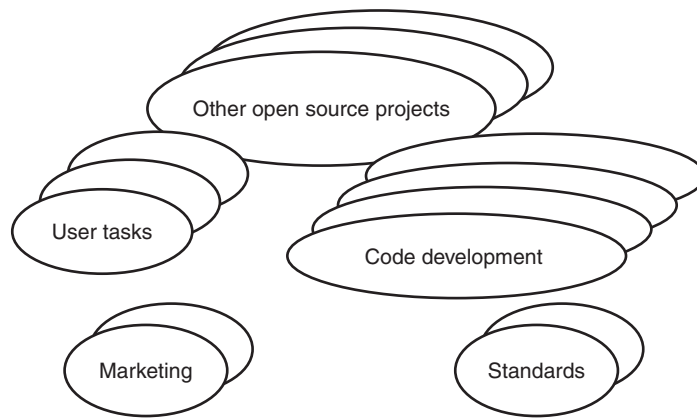


FIGURE 3.3 Communities built around common interests.

that are available for Linux—their focus is on what tools exist and how to use them. A third community might form around computer games available for Linux, with a subcommunity around a specific game such as Quake—this group would focus on exchanging tips, rumors of new games, and finding opponents to play with.

Each community will flourish or wither depending on how well its interests are met by the community resources. For example a community of newbies asking basic questions about how to use a piece of software will succeed only if more experienced users who can answer those questions are also part of the community.

In the course of a successful open-source project, different communities will come and go. New ones will spring up, grow, and possibly become dormant or die. As long as there are always some thriving communities, the larger open-source project can be considered alive and well.

Note that death of a community does not equal failure. Consider a community that arises to develop a new standard. After the standard it developed has been accepted by the larger Internet community, the community has achieved its purpose and is no longer necessary. If future revisions to the standard are called for, the community might be resurrected.

EXAMPLES OF MULTIPLE COMMUNITIES

To make this more concrete, let's look in depth at some of the different communities within two Sun-sponsored projects, Jini and NetBeans.

Jini

Jini technology is a simple distributed computing model based on the Java programming language developed by Sun. Among other things, it was intended as a model for services—small bits of functionality—to discover each other dynamically and create a network of interoperating program parts. These services could be housed within devices—physically separate computing platforms as far as Jini is concerned.

For Jini to succeed, it was clear that the underlying Jini protocols and infrastructure would need to become pervasive, and to accomplish that would require a strong community of participants and partners. Moreover, Sun did not have the expertise to define Jini services in areas such as printing, digital photography, storage, white goods, and the many other potential markets for products that would benefit from being network enabled.

Within the Jini Community that has developed since the core code was released, there are many separate communities focused on creating Jini services in different application areas, as shown in Figure 3.4. A small number of developers care

28 Chapter 3 *What Is Open Source?*

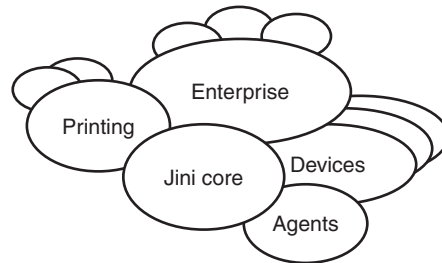


FIGURE 3.4 Different Jini code development interests.

about further developing the core Jini code. Others care only about areas such as printing or enterprise applications or building networked appliances that use Jini to connect to services. People working in one application area often have little in common with those working in another area. For example, those people working on creating services for appliances in the home have a very different set of concerns from those using Jini to connect legacy enterprise applications.

A similar situation exists in any large open-source project. In Apache, for example, there are smaller subcommunities focused on the Apache HTTP Server, the Apache Portable Runtime, the Jakarta Project (which includes major efforts such as Tomcat, Struts, and Ant), `mod_perl`, PHP, TCL, and XML (with subprojects such as Xerces, Xalan, Batik, and Crimson).

In addition to communities focused on developing code, other Jini-related groups have formed around interests such as helping beginners learn Jini, Jini in academia, and even Jini marketing and Jini business development. As of spring 2004, there were over 150 projects (although some of them seemed to be no longer active).

A characteristic of the Jini community that is not typical of other open-source communities is its elaborate governance mechanisms. Membership is divided into bodies consisting of individuals and companies, called houses. In the individuals house each person has one vote, and in the corporate house each company has one vote; and any major decision requires both houses to agree. There is also an appeals board (Technical Oversight Committee) for which the individual house, the corporate house, and Sun Microsystems each selects three members.

NetBeans

NetBeans is a modular, standards-based *integrated development environment* (IDE) that software developers use to create and modify applications written in Java. An integrated development environment is a software bundle consisting of a text editor (for creating and modifying software source code), code-building

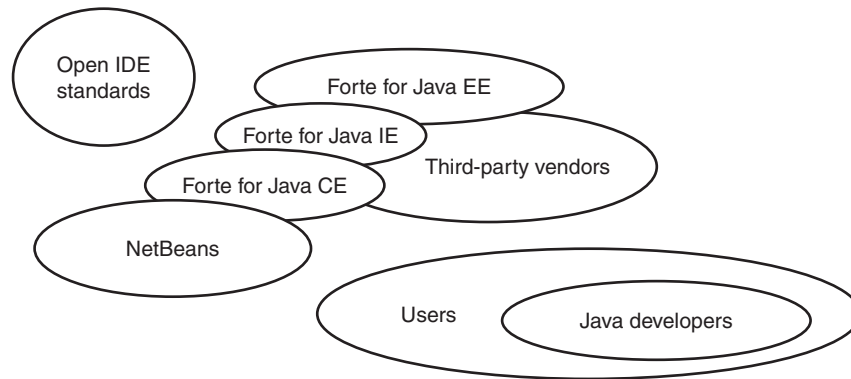


FIGURE 3.5 Multiple NetBeans products and uses.

automation tools (for combining source code components into a whole application) a compiler (for preparing source code for machine execution), a code execution platform (an interpreter or runtime environment to run the code), and a debugger (for locating and repairing coding errors). NetBeans supports a wide range of features, from JSP development and debugging, to integrated CVS support and beyond. All of these pieces of functionality are implemented in the form of modules that plug into the NetBeans core. As an open-source project, the basic version of NetBeans is available free to anyone wishing to use it. Sun's main interest in NetBeans is to bring developers to the Java platform and make them successful.

Figure 3.5 diagrams the various communities involved with the NetBeans project circa 2002. In addition to those directly involved with developing the NetBeans source code, there are three additional groups at Sun working on the various editions of Forte for Java: *Community Edition* (CE), *Internet Edition* (IE) and *Enterprise Edition* (EE). (*Note:* In late 2003, the Forte for Java product line was renamed as Sun Java Studio.) The last two editions are for sale products that include additional proprietary modules. There are also various third-party companies developing NetBeans modules that are also for sale.

There are several distinct user communities. First, there are those developers using NetBeans as an IDE to create Java applications—their concern is how to best use NetBeans to write Java programs. There is also a geographical information system (GIS) built using the NetBeans framework—users of that application are not involved with any programming; they want to discuss issues around using GIS. When the NetBeans IDE is enhanced to handle languages in addition to Java, such as C/C++, then there will be a new user community of C/C++ developers.

30 Chapter 3 *What Is Open Source?*

Note that in the diagram in Figure 3.5 the oval labeled *Users* should be several orders of magnitude larger than all of the other ovals; the users number in the tens or hundreds of thousands, whereas the people modifying the NetBeans code are in the hundreds.

For NetBeans to be a success, all of these communities need to be successful. Each must grow for NetBeans to prosper.

NetBeans provides another example of how an open-source project can involve multiple communities, in this case three very different cultures:

- NetBeans
- Forte
- Sun

At Sun, the NetBeans open-source project arose from the simultaneous acquisitions of NetBeans, a small startup company in Prague, Czech Republic, and Forte Software, a company based in Oakland, California. These two companies joined Sun's existing Tools organization, bringing together three companies, with three different corporate cultures, sets of values, day-to-day processes, and business goals. This was a challenging acquisition integration task.

Unifying three different corporate cultures and day-to-day processes would be challenging enough; however, the bigger challenge may have been that the Tools organization was now spread across three different sites (Menlo Park, Oakland, and Prague). Recall that distributed development is fundamental to the open-source methodology. It is simply assumed. So the open-source methodology represented both a natural solution for the Tools organization's physical distribution and a neutral fourth methodology to bridge the differences in corporate cultures and processes.

Other projects

Note that the users for both Jini and NetBeans are still mainly programmers. If we look at a project such as OpenOffice.org, which is aimed at creating an office suite for anyone to use, then the diversity of the communities becomes even clearer. OpenOffice.org is the open-source project based on Sun's StarOffice product, a multiplatform office productivity suite that includes the key desktop applications, such as a word processor, spreadsheet, presentation manager, and drawing program, with a user interface and feature set similar to those of other office suites. OpenOffice.org has groups focused on creating better user documentation, marketing and promoting OpenOffice, and usability issues. The basic community of people using OpenOffice has split to include (as of

March 2004) new groups for native-language discussion forums in over two dozen languages, including Arabic, Chinese, Dutch, French, German, Hindi, Italian, Japanese, Russian, Spanish, and Portuguese.

Hewlett-Packard is the central member of a printing community for Linux.¹⁴ IBM is the primary sponsor of an open source project for a Java-based universal tool platform and IDE called Eclipse,¹⁵ which, as of March 2004, spun off from immediate IBM control to become a nonprofit foundation. Within Eclipse, there are communities for such things as the basic platform, the Java IDE itself, for a variety of tools, for AspectJ, and for numerous other plug-ins and technologies. Each has its own mailing list, and there are a variety of newsgroups. The Eclipse project is discussed in depth in the section “The Eclipse Story” in Chapter 4.

LOOKING BEYOND THE CODE

An open-source project has many different communities, each with a different purpose. For a successful project, each of these communities must be nurtured and helped to grow. In a successful community a vocabulary might spring up that is derived from the project’s technology, application area, and existing culture. Such a community will come to resemble a long-existing club with its own phrases, in-jokes, rituals, and customs—an astute creator of such a community will know this and will help the community grow these aspects. A less astute one will focus on the code, probably leaving out vital potential members of the community.

One way to make the focus go beyond the code is to actively make roles for nondevelopers such as user interface (UI) designers and documentation writers. For example, NetBeans has a communitywide mailing list dedicated to the design and discussion of UI issues. There is also a NetBeans project focusing on UI development and issues, plus a process for other developers to get UI design assistance from the members of the UI project. When the NetBeans project first started, there was a hesitancy to add nondeveloper roles because this wasn’t something that the high-profile open-source projects such as Apache or Linux did. Now the community values the work of the UI group. A similar example is the recent addition of a usability group as part of the GNOME project—its work has been welcomed by the larger GNOME community as something long needed.

There are notable examples of communities that work together in an open-source-like way but do not involve software source code. One of the most

14. <http://hp.sourceforge.net> and <http://sourceforge.net/foundry/printing>

15. <http://www.eclipse.org>

32 Chapter 3 *What Is Open Source?*

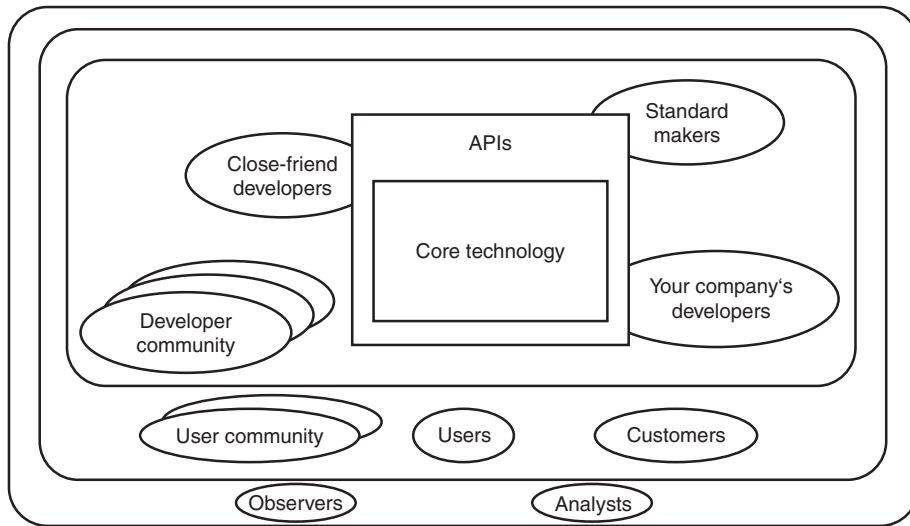


FIGURE 3.6 Possible constituencies.

interesting is Wikipedia.¹⁶ Wikipedia is an encyclopedia contributed entirely by volunteer efforts, using WikiWikiWeb technology at its base. In only 3 years, the online community has written over 230,000 entries. (Wikis are described in section “The Wiki-WikiWeb” in Chapter 8.)

Once companies involved with open-source projects realize that all of these other communities exist, they can consider business reasons for interacting with them—informal channels to customers, collaborations, and sources of innovation, for example. By cultivating new communities around user interests, such companies can work to ensure the success and growth of the underlying open-source project.

As you read the next chapter and think about your project’s business goals and business model, you need to consider how you will involve the various groups that have an interest in your project, as in Figure 3.6. Your business model must include a community focus.

The Secret of Why Open Source Works

Open source works when a group of people all embrace a set of shared goals and establish a community based on mutual trust. All three factors—enough interested people, shared goals, and trust—are required; if any one is missing, the project will fail.

16. <http://www.wikipedia.org>

SELF-ORGANIZATION THROUGH SHARED GOALS

Open-source projects often have hundreds or even thousands of people participating in their development. Yet there is no formal management structure to coordinate their many and varied activities.

The traditional approach to managing a large group of workers has been to establish a strict hierarchy of managers controlling the activities of the people below them. Such large command-and-control organizations have been the norm in business and the military from the Roman legions to General Motors. Such organizations pay a big cost in the number of people needed as managers, the amount of communications they require for coordination, and the rigidity of their response to novel situations. A big problem is that people lower down in the organization rarely have any idea how what they are told to do corresponds to the larger strategy. In fact, according to studies reported in the book *The Strategy-Focused Organization* by Robert Kaplan and David Norton, over 90% of companies fail to implement their strategy largely because the strategy is not communicated properly to the people in the company.

At the second Jini community meeting, Lieutenant General (Retired) Paul Van Riper, the former commanding general of the Marine Corps Combat Development Command, spoke on how the Marine Corps had made a shift from top-down control to policies based on self-organization for use in combat. For example, instead of commanding a platoon to capture a hilltop, they might now be instructed to set up an observation post on a hilltop in order to view enemy troop movements along an important roadway. If the initial hill is defended or inaccessible, the platoon can choose a neighboring hilltop that also satisfies the goal instead. Including the strategic purpose as part of each order results in a much more flexible and responsive organization. A similar organizational shift in the business world was presented by Tom Petzinger at the first Jini Community meeting—and is the topic of his book *The New Pioneers*.

Open source takes this self-organization process several steps further: Rather than someone high up in the organization setting out the goals, the actual people using and developing the software discuss on public mailing lists or newsgroups what improvements are needed. People propose features or capabilities that they want to see added—usually because they want to use them themselves. If enough other users and developers agree that the feature would be useful, then some of them may volunteer to write the code to implement it. However, if interest is too low, nothing may happen.

Sometimes a single developer will care enough about an idea to code up a prototype of it, and, if enough people try the prototype and like it, it may become part of the official project or at least be kept as an option for those who want to use it. Note that this type of exploratory development is a

34 Chapter 3 *What Is Open Source?*

common aspect of open source. It is a way of introducing new goals into the community through demonstration rather than just conversation. Some may complain that this is not efficient—the developer's effort is wasted if the innovation is rejected—but, as with evolution, efficiency is not the point. That developer may have had no interest in working on anything else aside from the innovation that was rejected and would not have worked on other parts of the project. So the work did not take away from existing efforts—although if the work had been adopted it might have meant that some existing modules would have been replaced. Also, even if a particular contribution is not incorporated into the project directly, the ideas behind it may very well find their way into later contributions.

Although the overall open-source project may have hundreds of participants, there always is a basic core team, usually consisting of fewer than 10 developers. With that small a group, informal communication suffices to coordinate development activities. In fact, the team works pretty much like any proprietary software development team does. Two main differences are that the open-source core team members are almost always geographically dispersed—so they communicate mainly via email and chat—and that they get immediate feedback from the rest of the community on their decisions.

For those larger open-source efforts (e.g., Linux, Mozilla, NetBeans, and Apache) that have many more dedicated developers, the work is divided into separate modules, each with its own small core team. This is a good example of Conway's Law—the architecture of the code follows the structure of the organization. This tendency to keep the code modular gives open-source software a flexibility and adaptability that is often lost in the more monolithic code produced by proprietary companies.

This pressure for modularization also can be seen in the evolution of project mailing lists: Whenever the traffic on a list or newsgroup gets too great, the community breaks the discussion into subtopics. The separation of the code and the associated discussions makes it possible for people with limited time to follow and actively participate in the project.

The project mailing lists also show self-organization in how questions get answered. For conventional products, companies employ people whose job is to answer customer questions. Open-source projects do not have such people, and at first glance we might expect that the main developers would be overwhelmed trying to answer questions from the thousands of users. This isn't what happens, however. Instead, the more advanced users answer questions from the intermediate users, who in turn answer questions from the novices, leaving only the extremely sophisticated or difficult queries for the developers to reply to.

Another way open source self-organizes is seen in how bugs are identified and fixed. The entire community is constantly using, and therefore testing, the

software. Each member shares the goal of wanting to improve it and so will report problems, often locating the code causing the bug and including a possible fix. This full testing coverage and repair takes place with no coordination required and minimal core-team overhead.

Having common goals brings people into the project and through group discussion these goals are refined and developed. However, as the project goes forward, if part of the community has goals not shared with the rest, then the project may fork. For example, when some members of the NetBSD community decided that security was a major goal that wasn't being given enough emphasis, that group decided to create a new project, Open-BSD, that had as its main goal the building of the most secure Unix-like operating system.

TRUST ENABLES COOPERATION

The sharing of goals creates a reason for people to participate in the open-source project. To create a true community requires trust—trust that your contributions will not be scorned and that you won't be made a fool of or taken advantage of. When participants show each other mutual respect, it becomes possible for them to cooperate. If discussions degenerate into flaming and put-downs, if suggestions and contributions are laughed at, if decisions are made in an arbitrary way, then most folks will go elsewhere and not put any energy into the project.

A classic example of how many open-source projects work to build trust is shown by how a new developer acquires the right to modify the project's source code. When new developers join a project, they need to earn the respect of the current developers. They do this by submitting bug fixes and modifications and by participating in group discussions. Only after demonstrating that they can act responsibly will they be granted check-in privileges. Thus, developers know that the code they write can only be modified by people they trust; their code is safe from anyone who would arbitrarily or ignorantly change it.

Another concern people have is that they be treated fairly. Before investing effort into a project, most people want to be assured that they will have a voice in any major decisions. When decisions flow out of community discussions and represent consensus that reinforces the sense of community, people will be invested in carrying out the necessary work. If decisions seem arbitrary or go against their interests, then people have no motivation to participate and are apt to leave. Many open-source developers worry that a company will decide to sponsor an open-source project because it plans to take the community's work, package it up in a proprietary product, and deny the community access to it. If these fears are

36 Chapter 3 *What Is Open Source?*

not laid to rest and the company doesn't earn the community's trust, then it will be hard to attract outside developers to work on the project and it is even likely that a competing open-source project will be created—possibly starting with the very code that the company donated.

It is important to remember that open source is based on a gift economy: Participants trust that the gifts they give to the project will be returned in kind by other community members. It's more like a family than a consortium—individuals create relationships with other participants though continuing interactions; the relationships that hold the project together are person-to-person, not company-to-company.

ACHIEVING CRITICAL MASS VIA THE INTERNET

The Internet is a major enabler of open-source projects. It helps people living all over the globe to find others with common interests. Through email, newsgroups, and the web, these people can easily communicate, share code, and coordinate their activities. Without the Internet, it would be vastly more difficult, if not impossible, to run a successful open-source project.

An open-source project requires people, both users and developers. Many open-source projects start with a developer or two who have a program that they feel others will find useful. They can advertize their application through newsgroups and mailing lists, and they can make it available on relevant websites. Through the use of search engines and special-interest lists, people anywhere in the world can then locate and try out programs they might be interested in. People who like a program will spread the word. This makes it possible for groups to form based on very specialized interests—with the whole, networked world to draw on, it's hard to think of a topic that won't draw a quorum.

After finding a program they like, people are able to provide feedback to its creator via email and they can also discuss issues of using it with other users. The ease with which new mailing lists, newsgroups, and websites can be created makes electronic discussions both quick and simple. Email discussions among a large group of people can be easier both to follow and to separate by topics than face-to-face discussion. Mailing lists also allow the group to coordinate future development.

While the Internet makes it easy for new individuals to find an open-source project, whether or not they stay and participate will be determined in large part by the project's culture—is there a sense of community and trust, or instead is it cold, cliquish, and unfriendly? Unlike the business world where people assigned to a project are forced to work together, with open source, interested potential contributors who do not engage with the core developers—because

of having a different vision or due to incompatible work styles—will probably go elsewhere.

OPEN SOURCE IS AN EVOLUTIONARY PROCESS

The open-source process involves small, incremental improvements to some artifact, usually the source code for a computer program. It is essential that the starting point be a useful artifact, such as a working computer program. If the program is incomplete and does not do anything, then a small improvement will result in a similarly incomplete, broken program—so there is little motivation or satisfaction in making a small change to it. To finish the program and make it useful would require a large amount of work and initially only the originator has such a strong commitment.

A direct consequence of this is that an open-source project that starts around a great idea, but with no code written, should not expect the community to create a working application. The community can only be expected to help develop and refine the idea through discussion. If there are one or more developers already committed to writing the initial version, then this feedback on what the application should do can be invaluable—although without a working prototype it can also be ungrounded and too blue sky to be practical. The main point is that if you start an open-source project without a working prototype do not expect the community to write the initial code for you. It just won't happen.

It may help to think about this in terms of biological evolution: Dead matter does not evolve;¹⁷ only living organisms can evolve. They do so as a result of random small changes. Some of the changes are improvements, whereas others are neutral or harmful to the organism. The environment then selects which of these changes are retained and which are forgotten, and these selections are passed on through reproduction.

For open source, every new idea posted to the project's mailing lists and each contribution of code is a potential change. With new ideas, the community acts to select which are to be discussed further and possibly become part of the project's goals. For code, the module owners are the ones to select which changes are accepted and added to the project's code base. Note that the module owners cannot dictate what other developers work on—in some sense, incoming contributions really are random.

Most changes will be small bug fixes or minor improvements, but occasionally someone will contribute ideas or code that is a major change, possibly taking the

17. Dead matter can change, but because it cannot reproduce dead matter cannot propagate beneficial changes.

38 Chapter 3 *What Is Open Source?*

project in a new direction. As with biological evolution, these large jumps cannot be predicted. They are also experiments that may or may not succeed. Just like life, open-source projects must embrace these opportunities to remain vibrant.

Evolutionary processes tend to be profligate in their use of resources—they are not what a manager would consider efficient. However, through large-scale parallel exploration of possibilities, evolution discovers wonderful innovative solutions. Open source combines the directed efforts of proprietary software development, in the focused work done by the core team, with the open-ended contributions made by volunteers.

CO-EVOLUTION OF SOFTWARE AND COMMUNITY

In an open-source project, software building and community building are intertwined. As the software matures, the community needs to keep up with it—the principle of slowly dawning insights applies to both activities.

An example is the Apache Incubator Project, which was created in October 2002. It provides an entry path into the Apache Software Foundation for projects and codebases whose owners wish to become part of Apache. The Incubator also provides documentation on how the Foundation works and how to get things done within its framework.

The Incubator is a community structure that was developed after the code base became large, sophisticated, variegated, and widely adopted. This required creating a controlled and self-explanatory mechanism for joining the community.

The Apache Software Foundation has apparently discovered the work-in-progress effect and is using it explicitly. At the top of the home page for the Incubator¹⁸ it says the following:

Notice: This document is a WIP (Work In Progress).

Variations on Open Source: Gated Communities and Internal Open Source

Sometimes a company is not willing or able to relinquish all control of the source code it has written. It may be that sale of the software is too important a part of their business. Or they are unwilling to give up the intellectual property (IP) used in the software—or they cannot because they use another company's IP that they only license. Whatever the reason, they choose not to open source their code. There are two important variations on open source that they can consider in order

18. <http://incubator.apache.org>

to get at least some of the benefits of collaborative development. The first is to share only with people or companies that agree to a license that lets them see the source code but that strictly limits what they can do with it. The other alternative is to keep the source code totally within the company, but allow access to it by their developers who are working on other projects. Both options reduce the size and scope of the potential community, but can still provide additional value.

GATED COMMUNITIES

One of the hallmarks of open source is that anyone can redistribute the code, with or without changes, to anyone else. This is not so when the source requires a special license that restricts the distribution only to other licensees. For example, from 1975 until 1992 anyone wanting to access the source code for the Unix operating system was required to purchase a license from AT&T. Many organizations relied on the Unix sources distributed by the University of California at Berkeley, but they all were supposed to have an official Unix license from AT&T. It was only after 386/BSD was released in 1992, followed by releases by both the NetBSD and FreeBSD projects in 1993, that source code for a full Unix system became freely available as open source.

A source license can be used to create a gated community: Anyone agreeing to it is in the community, and anyone who does not is left outside the gate, unable to see and use the source code. This can be attractive to the company that wrote the source code because it can still sell the software and retain all of its IP. Whether this is attractive to outside developers depends on the other license terms.

The most restrictive source licenses only allow a licensee to look at the source code—a licensee cannot modify or redistribute it. Even this little access can be useful if you are building other software that must interoperate with the licensed software. The source code essentially provides additional documentation and can aid debugging. This is what Microsoft offers with its Shared Source program for its Windows operating system. Those few companies that qualify can use the source code to help them better understand Windows in order to debug or tune their own hardware or software. A less restrictive source license might allow a company to modify the source and use it internally, but not distribute it to anyone.

Only when the license allows redistribution is an open-source community possible. For example, Sun's Free Solaris Source License Program allows anyone who has signed the license to view and modify the Solaris source code. Licensees are allowed to distribute their changes only to other licensees via a Sun secure website that includes mailing lists for discussing the source code.

40 Chapter 3 *What Is Open Source?*

Licenses of this type often make a distinction between commercial and research use. Commercial use is when a company takes the source code and modifies it to create a product it then sells or uses internally for a production system. Such commercial use may require an additional license, possibly including royalty payments to the original author of the source code. Research use involves no sales or other commercial gain, and the distribution of binaries to anyone may be allowed.

The Java community is a good example of a gated community. It is an open organization of international Java developers and licensees whose charter is to develop and revise Java technology specifications, reference implementations, and technology compatibility kits. Java technology was originally created by Sun Microsystems and released under the Sun Community Source License (SCSL). Decisions are made using the Java Community Process (JCP), which has evolved from the informal process that Sun used beginning in 1995 to a formalized process overseen by representatives from many organizations across the Java community. The original reason for using a gated community was to maintain the compatibility of different Java implementations—write once, run anywhere. Now that Java is established and seen as a standard, the JCP is beginning to allow true open-source Java development.

The Jini project is another example of a gated community. It uses a variant of the SCSL that allows free commercial use and has a more democratic governance structure.

In fall 2000 Hewlett-Packard's printing and imaging division launched what it calls the Collaborative Development Program (CDP), a secure web-based development environment that links HP's worldwide employees, business partners, and customers to collaborate on software development projects. As of early 2002, the program hosted over 350 projects and 3000 users, approximately 10% of the users are external to HP. Forty-five external companies are developing projects with HP using CDP.

Some of the benefits of joining a gated community include using the source code as additional documentation, enhanced ability to find and fix bugs, ability to make local modifications, and ability to obtain and share modifications with others. The disadvantages can include the tainting of your developers by their seeing proprietary code, licenses that do not allow modifying the source code, and licenses that do not allow you to share changes with others.

The company that originally created the software can benefit from additional sales due to the extra value customers perceive from having access to the source code, assistance in finding and fixing bugs, customer porting of the code to additional platforms, improvements that can be distributed to other customers, and receiving better feedback from customers.

INTERNAL OPEN-SOURCE

The source code for most proprietary software is usually seen only by the team of developers assigned to work on it. Some companies limit which employees are even allowed to look at a program's source code, and most companies severely control who can modify source code. Some companies do have code reviews where people not on the team look at the code, but their involvement is generally limited to a critique of the code.

It can be to a company's benefit to open up the source code to everyone within the company. Access to a product's source code provides documentation to those developing other products that must interoperate with it. It can help in testing and fixing bugs. It can facilitate code reuse. In short, all of the same benefits that outside companies can get by being able to see the source code are available with internal open source. Even when sharing totally within a company, proposed changes must still be approved by the project's core team or those people who have earned their trust. We refer to such projects as internal open source, but others sometimes use the term *corporate source*.

Sharing source code within a company is much simpler than sharing it with those outside. Internal use requires no software licenses, just putting the source on some internally accessible file server. Of course, so far we're speaking only of the technical aspects of sharing—it may take a very big shift in mindset to open up a project's code to developers in other parts of the company. In fact, one of the biggest benefits might be increased communication between different parts of the company.

The focus on creating communities is a major difference between internal open source and other programs many companies have to encourage software reuse. Software reuse has at its core the idea of a library of reusable software components maintained by a code librarian. The code is usually seen as being static—contributed by a developer on one project for use on unrelated other projects. Software reuse projects often fail by not addressing social and political issues inside the organization. Internal open source must also overcome similar social and political obstacles, but they are faced more directly in working to build a community of developers and users to collaborate on an application that the community is all interested in moving ahead. If your company has a software reuse program, to make it more successful consider using open-source principles to build communities around the various components and frameworks.

Because internal open source takes place totally inside a company, there is no way to know just how many companies are applying open-source principles to their internal development process. Both VA Software, with their SourceForge product, and CollabNet, with SourceCast, are companies committed to open

42 Chapter 3 *What Is Open Source?*

source that make their living by selling collaborative software development tools to other companies. Although many of their customers may only be seeking assistance for geographically distributed project teams, others are no doubt taking advantage of the ability to involve people outside of a project in its development.

Hewlett-Packard started a Corporate Source Initiative to use open-source practices internally in June 2000. One of the major goals of this program is to increase the technology transfer from HP Labs to the rest of the company. By using the community building aspects of open source, HP hopes to build a stronger community within HP Labs and then extend that community to include other developers in the rest of HP who are working on products, infrastructure, and corporate operations. By early 2002 it had about 1500 registered users working on about 30 projects, all of which were research projects not tied to any HP product. Researchers at HP Labs have written about a variable approach to open source they call *progressive open source* (POS) to describe the range of ways a company can use open-source development methods: traditional open source, gated communities, and internal open source.¹⁹

IBM uses tools from VA Software—SourceForge Enterprise Edition—to host an internal-project, open-source site called the *Internal IBM Open-Source Bazaar*, where any team can do open-source development within IBM. The tool provides a source-code repository, mailing lists, source-code control mechanisms, and license management. The site supports hundreds of projects and thousands of developers working on open-source projects that IBM doesn't want exposed to outside parties. IBM is one of about 50 companies that use SourceForge Enterprise Edition for their internal open-source projects.

Sun Labs set up a very similar program inside Sun called OpenProject, starting in December 2001, with the twin goals of helping with technology transfer from Sun Labs to the rest of Sun and providing a home for small, unsponsored projects to encourage innovation. In May 2002 the scope of OpenProject grew with the addition of a number of groups from Sun's Enterprise Services organization that wanted to build communities around the development and use of internal Sun tools. These tools include the various applications that are used internally by Sun field engineers and customer support to install, test, and maintain Sun computers. The aim is to enable the worldwide shared development of these tools, as well as to create forums where Sun employees can locate the tools they need and discuss how to best use them. By March 2004 there were over 800 registered users of OpenProject working on over 300 hosted projects, most of which were internal tools.

19. Jamie Dinkelacker, Pankaj K. Garg, Rob Miller, and Dean Nelson, *Progressive Open Source*, HP Labs, 2001.

To manage and guide this internal tool development effort, a special Technical Council was created with representatives from organizations throughout Sun. This Technical Council is responsible for locating existing tools and working with their developers to contribute their source code to OpenProject, for identifying gaps in the existing tools and developing new tools to fill them, for pointing the community to what is cool and identifying the best-of-class tools, and for encouraging groups and engineers throughout Sun to work together.

Internal open-source projects need to be nurtured just as all open-source projects do. In fact, they may need even more support from managers and executives because the people involved in them do so as part of their job and, although the project may benefit the company as a whole, their involvement can be seen by their immediate manager as taking too much time from their assigned job.

Done correctly, internal open-source allows a company to leverage the work of all of its employees, to eliminate duplicate work, and to encourage innovation. The same open-source principles come into play whether the potential community is the entire Internet or just a single company's intranet.

Open Source: Why Do They Do It?

One of the first questions a dyed-in-the-wool business person will ask about open source is why in the world people would volunteer to do something that they could be paid to do. Numerous explanations have been put forward, including the following:

- Need for the product—in order to create, customize, or improve a product or feature. This reason dominates the decision-making process for all early participation in both open-source and gated-source projects, but, as time goes on, continued participation is based on other things for open-source projects whereas a need for the software continues to prevail as motivation for gated-source projects.
- Enjoyment, fun, and desire to create and improve—because they enjoy it, find creating or improving software creative and interesting. This is the primary reason people continue in open-source projects for the long term. Such people tend to scan the email archives, bug reports, and feature-request lists to find things that catch their eye, things that are challenging or represent an area they want to learn about.
- Reputation and status—in order to build or maintain reputation or status within the community.
- Affiliation—in order to socialize or spend time with like-minded individuals.

44 Chapter 3 *What Is Open Source?*

- Identity—in order to reinforce or build a desired self-image.
- Values and ideology—to promote specific ideals, such as the free software philosophy.
- Training: learning, reputation outside the community, and career concerns—to improve their skills, with the belief that such improvement will lead to a better job or promotion.
- Fairness—to pay off the debt they owe from having used the software or received help from the community. For some, the bargain takes a long time to pay off.
- Hope of making things better—to find or create better solutions than those already in place.
- Feedback—to get comments on the work and how well they are doing as a programmer or designer. As with other creative activities, this is a driving urge. Here is what open-source expert Sonali Shah says,²⁰

... creative programmers want to associate with one another: only their peers are able to truly appreciate their art. Part of this is that programmers want to earn respect by showing others their talents. But it's also important that people want to share the beauty of what they have found. This sharing is another act that helps build community and friendship.

What Is Open Source?

Open source is about software source code, licenses, communities, culture, and distributed software development. Although open-source projects can provide plenty of benefits for companies who use it, open source is not something to do on a whim. Open source is based at least in part on a phenomenon called the gift economy, which on the surface seems at odds with corporate practices.

Nevertheless, marketing and innovation benefits, as well as clearly separating commodity efforts from value-enhancing ones, can make all the difference in a business climate that values carefully thought-out innovation.

20. *Community-Based Innovation & Product Development: Findings from Open Source Software and Consumer Sporting Goods*, p. 33.