Toward Taxonomy of Open Source: A Case Study on Four Different Types of Open-Source Software Development Projects

Kumiyo Nakakoji[1,2,3] Yasuhiro Yamamoto[2,4] Yoshiyuki Nishinaka[1] Kouichi Kishida[1]

[1]SRA Key Technology Laboratory Inc.
3-12 Yotsuya, Shinjyuku
Tokyo, 160-0004, Japan
+81-3-3357-9361

[2]Grad. School of Information Science, NAIST
8916-5, Takayama-cho, Ikoma
Nara, 160-0004, Japan
+81-743-72-5381

[3]PRESTO, JST

[4]Japan Society for the Promotion of Science

{kumiyo, yxy}@is.aist-nara.ac.jp {nisinaka, k2}@sra.co.jp

ABSTRACT

As more and more data and experiences on open-source software development are reported and shared among the community, a number of controversial arguments regarding the benefit, necessary support, key issues, and challenges for open-source projects have emerged. Many of these stem from the fact that the meaning of the term *open-source software development* (OSSD) has become unnecessarily overloaded . Different OSSD projects have radically different goals, motivations, development processes and communication styles. Accordingly, they benefit in different ways from being "open".

This paper presents our case study in which we analyze and compare four different types of OSSD projects carried out at SRA Inc., Japan. Through the case study, we have identified a wide range of variety among the software projects, all of which labeled "open source," and have found that there are at least three different types of OSSD: *archetype*, *security*, and *rapidness*. Each type has a different style of project management, development, communication, and evolution. This paper concludes by arguing that we need a taxonomy for the different types of OSSD projects in pursuit of technological and practical support for OSSD.

1. INTRODUCTION

Although more and more data and experiences on open-source software development have been reported and shared among the community [11, 13, 14, 15], there still have been a number of controversial discussions over open source software development (OSSD). Arguments include:

- whether OSSD produces more secure software than traditional development,
- whether OSSD really embraces evolvability of the software,
- whether OSSD results in better quality than traditional development processes,
- whether its development style actually adheres to more "cathedral" than "bazaar," and
- whether OSSD really helps community development.

Many of those controversial arguments, however, seem to stem from the overloading of the term, *open source software development*. In fact, there are many completely different development approaches which are labeled *open source software development* only because they produce open source software under open source licensing, such as GPL [6] or BSD's [2]. The term, open source, is thus overloaded, since it refers only to the property of the outcome; not necessarily to the process or to the project. Motives, processes, goals, may significantly differ for different open source software development projects. However, there has been little effort reported to classify different types of OSSD other than looking at licensing differences. We need a taxonomy for representing different types of OSSD so that we can better talk about what technological and institutional support are necessary, when a project should go or not go for open-source, and what we can expect for being open source.

In January 2001, the Information technology Promotion Agency (IPA) of the Ministry of Economy, Trade, and Industry (METI) of Japan, decided to conduct a survey on the current status of OSSD in the Japanese software industry. Our company, SRA Inc., was awarded the grant to conduct the survey, which identified different types of OSSD, and compared existing industrial and governmental support for OSSD in different countries. SRA is a leading company in the open source movement in Japan, who has been supporting the activities of the Free Software

Foundation (FSF) since 1987, and has carried out a variety of open source software development projects within its Open Source Business Division. This paper is based on a part of the findings from the survey, which is a case study over four different open source software development projects conducted within SRA, Inc., Japan.

The four projects we have looked at are (Table 1):

1. the *GNUWingnut* project, which provides support for a number of GNU software applications, such as GCC (GNU Compiler Collection) and GNU Emacs, for Japanese industries who need GNU software ported into their hardware platforms;
2. the *Linux Support* project, which offers support for Linux users as a master SI distributor;
3. the *SRA-PostgreSQL* project, which supports Japanese customers who use the PostgreSQL database, which is an open source database; and
4. the *Jun* project, which is a 3D graphic and multimedia application library for VisualWorks Smalltalk and Java.

Table 1: The Four OSSD Projects Studied

	OSSD Project	Domain
1	*GNUWingnut*	development environment
2	*Linux-Support*	operating system
3	*SRA-PostgreSQL*	database
4	*Jun*	3D multimedia library for Smalltalk and Java

Those four projects vary in their development styles, communication styles, management styles, evolutionary styles, and in their underlying philosophy.

We identified three major focuses: *archetype*, *security*, and *rapidness*. The primary objective of the first type of software is to serve as a reference model. The objective of the second type is to achieve rapid recovery. The third type's primary goal is to have timely development by distributing work among a community of users.

Through the case studies, we have identified a wide range of varieties among the software projects, which are all labeled OSSD. We argue that we urgently need a taxonomy and better names referring to different types of open source software development. Such a taxonomy, would enable more constructive, fruitful discussions on how and why open source is a promising approach for the development of a certain type of software, which of existing software engineering frameworks should be used to support open source software development, and on which aspects of open source software development cannot be supported within the existing software engineering approaches.

In what follows, we first describe a brief overview of each of the four projects we have studied. We then compare the four projects from different perspectives including uses and stakeholders, communication styles, evolutionary processes, and primary focuses. Based on the comparisons, we present our initial taxonomy for OSSD. The paper concludes with an illustration of how such a taxonomy can be used to better understand the effect of being open-source, followed by a discussion of future directions.

2. THE FOUR PROJECTS

This section first describes how we conducted the case study. We then present an overview of each project, describing what open-source software each project deals with, how each project does business with the software, and what benefit they realize from the software being open-source.

2.1 A Case Study

A survey was conducted by interviewing the project manager of each of the four open-source software development projects. During the interview, we asked questions including:

- what open-source software they are dealing with;
- how the development of the open-source software has been done;
- what communication media the developers use in the development of the software;
- how they do business with the open-source software; and
- what benefit they see by doing business with open-source software.

By addressing these questions, their answers often got expanded further, and these questions served as seeds for discussions, not necessarily obtaining answers to the specific questions. In addition to the interviews, we asked for their mailing-list archives and quantitative data related to particular aspects when necessary.

Note that the open-source software we describe in this paper reflects the views of those with whom we conducted the study . The views and opinions expressed by the project members who were interviewed may not be consistent with that of the core members of each project. For instance, we have interviewed the SRA-PostgreSQL project members at our company, but we have not interviewed with the PostgreSQL core development team

members. This case study is to report how the OSSD project members at a for-profit company view their OSSD, and how different types of OSSD results in different types of business projects.

2.2 The GNUWingnut Project

As the name states, this project deals with GNU (Gnu is Not Unix) software [7] developed by FSF (Free Software Foundation) [4]. The GNUWingnut project helps clients import GNU software programs onto their particular hardware platforms. GNU is a software project that develops a "free" unix operating system organized by Richard Stallman at FSF. For Stallman and FSF, programs are "scientific knowledge to be shared among mankind" [8]. That is, for them, software is knowledge developed by "highly trained professional programmers" and therefore to be shared among human beings in the same way as the knowledge medical doctors develop is shared through research papers and books. It is this spirit that makes their software *free*. They have been using the term "free" not to mean that the software is free of charge but the source code is free to view, modify, and distribute under the license called GPL (GNU Public License) [6] with the ownership notion called copy-left [10].

Although it is not our purpose here to describe GNU and FSF in detail, several interesting characteristics to note. Although it may not be explicitly stated, this view of programs as scientific knowledge has developed a culture where open-source programs need to be of very high quality; they want to develop the "correct" and "best" program for implementing a piece of functionality. Because it is to be good and to be shared among mankind, "democratic" decision-making and centralized control has been exercised. GNU software development teams observe strict coding rules and format guidelines [9] to make their software to be easy to be shared among mankind. Only one version of the software is allowed and variations and alternatives need to be integrated within the core version. All bugs found need to be reported so that the core members can fix them. Overall, control is very much centralized.

The main task of the GNUWingnut project is to help clients port GNU programs into their target machines. A typical case is that a hardware vendor needs to have GNU Emacs and GCC operate on their super-computer operating systems. This involves two types of work. The first type of work is to develop patch programs for the clients. Although the source code is available, many GNU programs are very large and complex and require substantial knowledge and experience to understand. The GNUWingnut project members offer such expertise, enabling clients to develop patch programs faster and better.

The second type of work, which is more interesting and possibly unique to GNU-related software development, is to help clients increase the quality of patches by revising and refining them so that they can reported back to the GNU core members. There seem to be three reasons why the clients need such help from the GNUWingnut project.

First, as we noted above, a GNU project wants to have a single version for a particular program and all bug fixes and updates need to be reported back in to the core development team. For instance, when a super-computer vendor develops a patch program of the GCC program for their super-computer operating system, this company needs to have this patch program reported back into the GCC core development team; otherwise they have to develop a patch program for every subsequent update of the GCC program. Second, as also noted above, GNU requires fed-back programs strictly adhere to GNU guidelines for coding, formatting, and documenting. Although most of these guidelines can be ensured by using appropriate "modes" in the GNU Emacs editor, it still requires expertise and skills in observing these guidelines. Third, there is a "cultural barrier" for Japanese programmers, which keeps them from directly communicating with the GNU core members through mailing-lists. Many programmers in Japan view the GNU core team as a group of super-programmers with highly respected skills, and want to keep a "respectful distance" from them. Some of the GNUWingnut project member have been closely working with the FSF members for the last decade, and they serve as the intermediary between the clients and GNU core members.

To summarize, because the source code is open, clients can use the software for their platforms by developing their own patch programs. On the other hand, the GNU software is under strictly centralized control, and those who create patch programs need to report their efforts back to the core development team. In addition to expertise required to understand the large amount of source code, this is where another type of expertise is necessary; to communicate with the core members, and to adhere to GNU coding guidelines. This is due to the fact that the GNU treat their open-source software as scientific knowledge to be shared among people. Such knowledge needs to be of very high quality and to be easily sharable among other people therefore needs strict guidelines.

2.3 The Linux Support Project

The *Linux Support* project at SRA Inc., provides user support for the Linux operating system, excluding the Linux kernal. We make this distinction because, similar to GNU, the Linux kernel development is under centralized control [15], while the remainder of Linux has been developed in the bazaar style with distributed control [21]. The Linux support

project is concerned with supporting the bazaar model, and accordingly in this paper when we refer to Linux we are referring to the portions of Linux outside the kernel unless specifically noted.

Contrary to the GNU programs, there have been multiple versions of programs for implementing a single functionality. No official centralized repositories have been developed for Linux OS peripheral tools, such as device and printer drivers. Each developer of a source code puts it on the Web; and Web search engines may find it when necessary. Because multiple versions for a single functionality exist all over the world (i.e., the World Wide Web), directory services are necessary to find necessary components. Also, there can be components that are not compatible with one another. Distribution packages have been developed to help customers find components that are compatible with each other.

O'Reiley specifies four types of business models that are possible by dealing with open-source [18]: (1) Support Seller, (2) Loss Leader, (3) Widget Frosting, and (4) Accessorizing. The Linux support project at SRA Inc., can be characterized as a "support seller.," that helps customers to identify and solve problems in the course of using Linux.. A typical task is to help clients find appropriate distribution packages and to customize software for their needs. Linux Support Project members are also asked to find up-to-date information on security and bug reports related to their clients' Linux programs, which are scattered all over the world on the Web.

Thus, what is most required for the Linux-support project members are (1) an ability to find necessary information, and (2) an ability to read and understand source code produced by other people. For instance, if a bug is found in a Linux program, a typical process taken by a project member is as follows:

1. first, read the newest version of the source code to see if the bug is fixed,
2. if not, then read the released version of the source code to see if the bug is fixed,
3. if not, then check a bug tracking report produced by the distributor if it reports the bug,
4. if not, then check related mailing lists to see if the bug is reported,
5. if not, then try to find Web pages that report similar bugs through the Web search.

When they find a newly-fixed program, they typically use the diff command to see how the bug is fixed, and apply the changes to the existing source code.

Interestingly, Linux Support members develop patch programs for their customers and fix bugs, but not necessarily report back to the developers. According to the project leader of the Linux support project, one obvious reason seems to be that the Linux customers do not care much about version updates. They stay with whatever the working version even if the version update is announced as long as the customer's system keeps working. When revision is unavoidable, they just want to re-install everything from scratch, instead of updating versions.

This is very different from the GNUWingnut project. In GNUWingnut, it is critical that the patch programs that have been developed and used at a customers' site are incorporated within the core GNU software version because otherwise they would be left behind; once incorporated, on the other hand, their drivers and interfaces will be taken care of by the GNU core development team. In contrast, Linux Support Program customers do not care about how a particular Linux program version evolves. If they find better ones, they will simply reinstall everything and develop patch programs from scratch.

2.4 The SRA-PostgreSQL Project

The *SRA-PostgreSQL* project deals with the PostgreSQL database [20], which is an open source database system originally developed as a research prototype . The system has evolved with an SQL interface and is now comparable with large-scale commercial database systems.

Being a database system, the robustness is a "must" for the PostgreSQL system. The system goes under a very strict centralized version control, supported by the core development team and the major development team, members of each of which are strictly controlled through the membership to specific mailing lists.

Democratic decisions regarding the development of PostgreSQL are made within the development mailing lists. Discussions are not so much on the implementation and source code, but as on a specification of the system because for a database system, the change in specification may affect the overall performance and quality of the system.

The primary task of the SRA-PostgreSQL project has been internationalization and localization. This has been done in four steps: first, the SRA-PostgreSQL project members have locally developed patch programs so that PostgreSQL can deal with the Japanese language. Second, they modified the patch programs so that they were able to deal with any two-byte code languages and adhere to the BSD licensing. Third, the patch programs were incorporated in the main version of the PostgreSQL system. Finally, the internationalized version has been disseminated internationally.

In addition to internationalization and localization, the SRA-PostgreSQL project helped Japanese clients to port the system to multiple platforms, and conducted testing and benchmarking for their clients. One of the SRA-PostgreSQL project members is a member of the major development team, and the project served as a representative agent in Japan, providing a Japanese ftp site to for bug fixes and collection and distribution of patch programs. Many Japanese customers used to have trouble finding necessary information because most information regarding PostgreSQL is in English. The SRA-PostgreSQL project helped the customers by translating information into Japanese and by serving as agents mediating communication between customers and he PostgreSQL development team members.

For PostgreSQL, the biggest advantage of being open-source is that source is kept open so that people can more quickly find what is wrong in the source --- bugs become shallower with many eyeballs [21]. In fact, not many people in the PostgreSQL user community contribute their source code. They do testing and finding bugs by using the source code, which is publicly available.

Another interesting aspect of the PostgreSQL project comes from the fact that the software is a database system. When reporting a bug, it is often necessary to use a specific set of data to reproduce the bug, since without the data it is very difficult to have programmers understand what is wrong and debug it. However, such data is often proprietary and cannot be made public. The customers therefore asks the SRA-PostgreSQL members to debug it, by making their data set available only to the project members. The SRA-PostgreSQL members then find what is wrong, develops patch programs and bug-fixes, and report feedback to the PostgreSQL core development team. Because robustness is a critical issue for databases, it is very important that bug reports and patches are fed back to the core development team.

2.5 The Jun Project

The *Jun* project at SRA Inc., deals with the Jun library, which is a VisualWorks Smalltalk and Java application library components on 3D and multimedia data handling [12]. Different from the above three projects, this project deals with the software which has been developed in-house. We have reported how the Jun library has evolved over the last five years and how centralized decision making and continuous evolution has been achieved [1].

As noted in [1], it is not only source code that has been used by the community, but also the underlying object model that has been used by the community. It has been served as a reference model in the development of 3D and multimedia handling, ensuring us with the highly advanced status in the software community; a type of the loss-leader business model [18]. Jun's evolution differs from other well-known open-source systems such as PERL [23], or Apache [3, 14]. Instead of a wide community of programmers each contributing a small part, almost all of Jun was developed by a small group of three to five programmers at a time. The development process is strictly controlled by the single project leader, who does both quality control and decision making in terms of which directions the project should evolve. Though the community did not provide much source code, it did provide feedback, feature requests and bug notices.

The business using the Jun library is primary the software development using Jun. Although Jun is an open source library and is freely available for other development organizations and developers, the library has become quite large, and expertise is necessary to be able to understand and apply it. The Jun project has an obvious advantage using the library. The project members have been asked to develop research application systems using Jun, and to provide consultation on the use of Jun as well as the use of underlying models.

3. COMPARISONS

The previous section gives unique characteristics of each of the four projects. This section examines the four projects and their open source software development processes from multiple perspectives, finding characteristics across multiple projects.

3.1 Uses and Stakeholders

Open source software does not necessarily mean that everybody who uses the software reads the source code. In fact, many users simply use the software and may not care if the source code is available or not.

Figure 1 illustrates what the uses of each open source software, and who serve as users, readers, and contributors. Users mean those who use the software. Readers refer to those who look at the source codes but not necessarily modify the source, or find bugs and report it. Contributors refer to those who actually write source code and contribute to the evolution of the software.

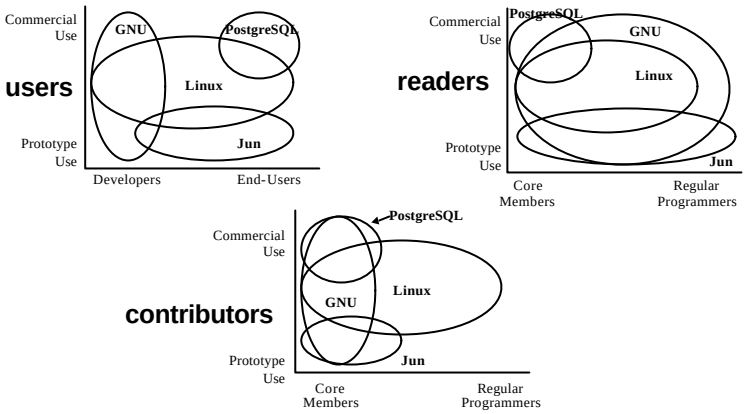

Figure 1: Uses and Stakeholders in the Four Projects

Because GNU programs are for development environments, many of GNU software users are developers. PostgreSQL is a database and has been used by end-users. Linux has been used both by developers and end-users as an operating system. Jun has been used as a prototyping toolkit and therefore used by developers, but also by end-users who uses application software developed on top of the Jun library.

GNU programs are kept open-source because they are scientific knowledge to be shared. Their source code is available so that everybody can look at, modify, and distribute them. By definition, therefore, readers of GNU span not only among core members but also among regular programmers. The Jun program also stands in the similar position. It has served as a reference model for 3D and multimedia modeling, having regular programmers as readers of the program. In contrast, PostgreSQL has source code open not for everybody to use it; but for having as many "eyeballs" as possible to have a look when something goes wrong. Consequently, the primary readers of the PosgreSQL programs are core members, and not so many regular programmers have the chance or the need to read the source code.

Who actually contributes to the evolution of the source code is very limited. Most of the open source software is contributed only by a small number of core members. This has been found true with major OSSD, such as Apache [14] and the Linux kernel [17]. Because there is little control over who does what in Linux, a wider audience will contribute to the Linux peripheral programs.

3.2 Community

All of the four OSSD projects extensively use mailing lists as a method of communication among development teams and among a community of users. GNU, PostgreSQL and Jun have official Web sites, where people can download software. However, these Web sites are not so much used as a communication medium. The PostgreSQL Web site provides an electronic bulletin board to report problems, but problems posted on mailing lists has more priority than information posted on the electronic bulletin board.

All the projects have core members who have leadership and responsibility in evolving and maintaining the program. All the projects, except the Linux support project, have another clear role division for those who play the intermediary between core members and regular users. Customers of open-source software often ask us for support to play this intermediary role, especially in the case of GNUWingnut and PostgreSQL.

Such roles are determined based on which mailing lists individuals belong to. Figure 2 illustrates how a community is formed for each OSSD project. In GNUWingnut, there are a few core members, who have leadership and responsibility for the project. Surrounding the core members are expert programmers, who report bugs, problems, and questions directly to the core member mailing list. Regular users communicate with those expert developers and feel less comfortable or feel not allowed to directly communicate with the core members. In Linux (excluding the kernel development, which is more like the GNU development), a small group of core members are responsible for the program, but everybody else is a regular user. There seems to be not so much two-way communication between core members of the Linux programs and users. Core members post updates and information on their Web sites; and users may find the information when necessary by browsing and searching the Web.

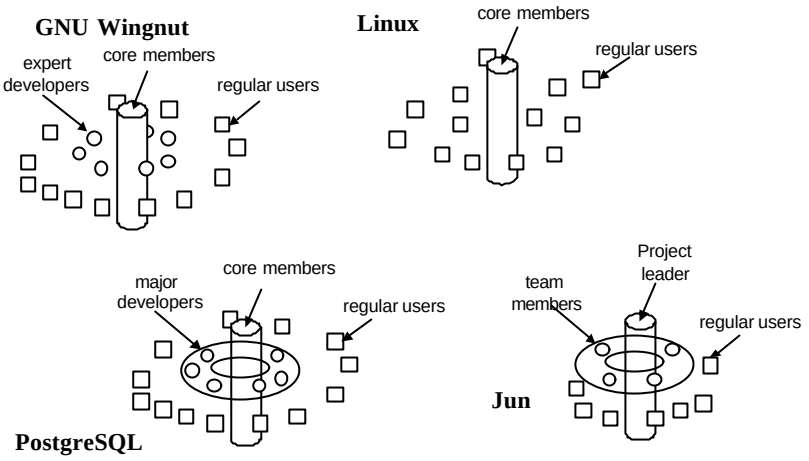

Figure 2: Community in the Four Projects

In PostgreSQL, there are six members in the core member mailing list, and about thirty members in the major developer mailing list. Who and how many should be on each of the two mailing lists are strictly controlled; it is usually voted among the current core members. In the Jun project, there is a project leader, and a Jun development team mailing list includes those who occasionally contribute source codes and bug reports. There is no mailing list for regular users, but they check Smalltalk Usenet newsgroups to obtain necessary information.

3.3 Evolution

Because each of the four projects has different objectives, the evolutionary path of each program also differs. Figure 3 gives a schematic picture of how evolution takes place in each of the four OSSD.

As stated above, GNU aims to have a single, clean, nice, well-written version of implementation for a single piece of functionality. When other people develop their own patch files for their platforms for the program, these updates need to be fed-back into the core version.

In Linux, on the other hand, there is much less motivation and encouragement for feeding back the updates. People develop patch programs, may or may not post them. Multiple versions for a single functionality are allowed, and there can be many branches evolving from the single version of a program.

In PostgreSQL, it is not so much patch programs that are fed-back into the core program; but new interfaces and new requirements for the database system are. New requirements emerge, members of the major developer team implement the requirements, then those implementations are incorporated within the core version when the core team member agrees to do so.

Finally, the Jun evolves also as a single-version tree. As is true of many OSSD projects, there are often branches of test versions created for internal usage [22]. When the project leader decides that it has been sufficiently tested, the tested version is released as a public version. In the case of Jun, every two to four versions are released public.

3.4 Summary of the Comparisons

By comparing the four projects, we have identified that there are at least three very different motives and objectives that OSSD projects may have:

- Dissemination of high-quality programs
- Fault tolerance by rapid recovery
- Timely development

The primary goal of both GNU and Jun is to have programs of high quality to be shared among people. By having the source code publicly available, they can use the code, modify the code to adapt to their specific needs, or to distribute the modified code under some licensing, such as GPL. This is the motivation for keeping the source code open.

In the case of PostgreSQL, because it is a database system, the security and robustness is the must. By having a source code open, they can use a community of users as many "eyeballs" to find bugs and problems as soon as possible.

Finally, in the case of the Linux operating system excluding the kernel part, developers develop what is necessary, for instance, a driver software for a particular printer. They put it on the Web with the source code open, so that if somebody needs such a program, they can use it. There is

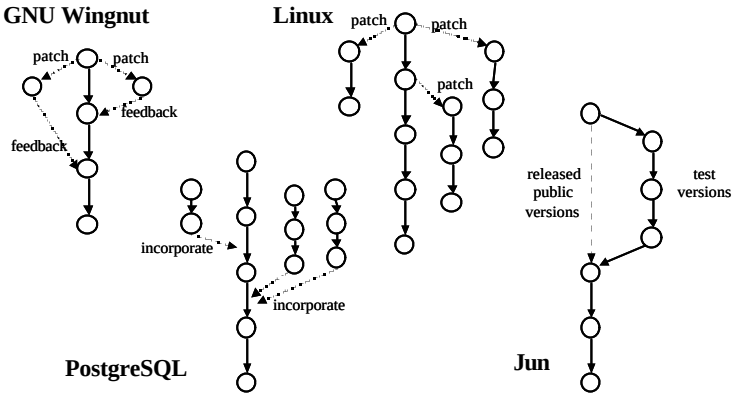

Figure 3: Evolutionary Process of the Four Projects

not so much desire of the developer to have everybody use his/her program and give feedback to the developer. Rather, they just make it publicly available. When Linux users need something, typically they first try to find it on the Web, wondering if there is anything available to satisfy the need. If not found, they find something similar, take it, modify it, and obtain what is necessary. Then they post it on the Web for further usage; in case somebody else might need it.

4. TOWARD A TAXONOMY

Based on the three points we have identified, we have constructed a taxonomy representing the three aspects: we call it archetype, security, and rapidness.

- *Archetype.* This type of software, represented by GNU software and the Jun library, is developed so that people can share the software and modify it if necessary. The primary objective is to save time and effort of software developers by not implementing the same functionality over and over again only because the source code is "closed." Being archetype, this type of software must be developed by expert programmers and must be of very high quality. Coding standard and programming styles are usually strictly observed. There should be a single archetype software program corresponding to a unit of functionality. Adaptations and bug-fixes carried out by the user community need to be fed-back into the evolution of the core version of the archetype software.
- *Security.* This type of software, represented by the PostgreSQL database, is developed as open-source so that it becomes fault-tolerant. It improves its security level by having many programmers examine its source code. As soon as someone reports a fault in its mailing list, PostgreSQL community starts looking for a cause and debugging. This type of software is usually very conservative against evolutionary changes.
- *Rapidness.* This type of software, represented by the Linux operating system (excluding the Linux Kernel, which is developed more like Archetype), needs rapid and prompt adaptation and modification when necessary. To take a hardware driver for the Linux OS, for instance, a programmer develops a driver by necessity and put it on the Web so that other people can take and use it. Users try to find program components on the Web when they face a need for the software. If they do not find one, they will develop it and share it via the Web. This type of software development is a typical bazaar type software development. Because there can be many alternatives and different versions for a single functionality, distribution packages are necessary to identify a typical set of program components chosen among a number of available programs.

We do not mean that these three aspects are mutually exclusive. In fact, many OSSD projects have more than a single focus, probably covers many focuses. However, an emphasis on a particular focus determines the project's management, development, communication, and evolutionary styles.

Table 2 summarizes our preliminary attempt to characterize each type. Because the primary goal for Archetype open-source software is to disseminate high quality software serving as a reference model, the control over the process and product needs to be highly centralized. It needs to be developed by responsible individuals, mostly by core members of the project. To maintain the quality, highly strict guidelines for coding, formatting, and documentation are enforced. Feedback is a must both on reporting bugs and informing of updates and patch programs. The single version needs to be maintained in order to serve as a reference model. Users of the archetype program are encouraged to access the source code, and to use , them, and redistribute the modified code. Learning is also

Table 2: A Taxonomy for OSSD

Focus	Objective	Control	Development-by	Feedback for debugging	Feedback on evolution	Versioning	Code access
Archetype	Reference model	Centralized	Core member	Must	Must	Single branch	All the time for modification and learning
Security	Fast fixes	Centralized	Core member	Must	Not necessarily	Single branch	When a bug is reported
Rapidness	Timely development	Distributed	Community	Not necessarily	Not necessarily	Multiple branches	Only by interested

encouraged by reading such well-crafted source code of this type.

For the same reason, Security-focused open-source also needs centralized control. For systems such as database systems, robustness is critical, and systems need to be well maintained and evolved by a set of responsible individuals including core members, and possibly well-trusted peripheral developers. Source codes are open so that widely available developers would help finding faults and fixes. Feedback on bug reports is a must to enhance the security, but evolution or modification of the program is not necessarily reported as a feedback. It might encourage the core members to refine system requirements specification, but the attitude towards change in this type of software is much more conservative and evolution may be slower compared to other types of open-source software.

Rapidness-focused open source, on the other hand, focuses on timely development and achievement of necessary functionality as a community. Control can be distributed as every individual would implement what he/she is interested in, and post it on the Web. It is the user's responsibility to find what is necessary, and whether or not trust the program. Feedback for debug and evolution is nice but not necessary. There is no guarantee that such feedback is taken into account by the developer, and it might not be trust-worthy feedback anyway.

Figure 4 illustrates which of the three aspects each of the four projects we have studied focuses on.

5. DISCUSSIONS

5.1 The Use of A Taxonomy

A taxonomy of OSSD styles allow us to articulate different causes and explanations for recognized properties of open source software. Let us take an example argument: *open source means high quality.*

For archetype open-source software, high quality means that the program embodies scientific knowledge to be shared among mankind. Such programs are created by highly skilled programmers, and released after rigid screening by peer expert programmers. The programs are then shared, followed, and used as a reference by follow programmers and users.

For security-focused open-source software, high quality means that bugs are found and eliminated because programs are exposed to "many eyeballs." The development process is highly controlled and a number of "code inspectors" are available as a community of users.

For rapidness-focused open-source software, high quality comes from the fact thatprograms are collaboratively developed by a community of users. The programs may be going through mutual critiquing, encouragement to contributions is given as a good "reputation." Tournament style evolution takes place and only good contributions survive over an extended period of time. On the other hand, this type of "high-quality" is rather opportunistic. This does not necessarily guarantee how good the programs are. This sense of high quality is very different from much more rigid ones for the archetype and security-focused ones. It is, therefore, pointless to compare the quality of GNU Emacs program with that of a printer driver for a Linux operating system posted on the Web both being open-source. Both are open-source, but they are very different creatures by its definition.

The three types of OSSD we have identified through our case study on the four projectsare by no means exhaustive. In fact, a large number of "typical" open-source software, such as Linux kernel, Apache and Mozilla, may not fall into any of the three types. They have centralized control over the development, but not as strict as GNU and PostgreSQL. They have quite clearly specified requirements, and each user of the community takes a small portion of the requirements and implements it. We may call this type "Task-Diffusion," but before we characterize this type, we need more case studies to identify common aspects among those projects.

5.2 Future Directions

Through conducting the four case studies, we have encountered commonly recognized issues to be addressed in pursue of the open source software development.

Support for documentation. All of the project team members mentioned that a large portion of the development of open source software involves documentation tasks. Documentation includes not only preparing install instructions and manuals, but also posting on the Web, updating links and related information, and announcing on the appropriate mailing lists. This type of task is often not taken into account as a development cost, and existing project still depends on programmer's spare time to prepare the document for their source code. Programmers are not trained for documentation and therefore are not necessarily the best people to ask to create such documents. In order to push OSSD, a role of documentation needs to be more emphasized and specialized personnel needs to be assigned to this type of the task.

Support for investigating software patent. Every open source software program is under some form of license. Licensing requires that algorithms used in the program are neither patented nor under any conflicting licenses. The task of investigating software patent is also left with a programmer's responsibility. However, it is not an easy task to do. Specialized support for investigating software patent is also necessary.

Need for code assessment by a third party. Now that source code is available, it is theoretically true that everyone can examines the source code to make sure that the program is written alright. In reality, however, it is not easy to understand a large and complex software program that is written by somebody else. Program reading requires different types of skills from program writing. It is necessary to have a sort of a third party, what we may call "software sommelier," who performs code assessment for a given open-source software program.

Learning how to read programs. Much of software engineering education has focused on teaching students how to write programs, but not how to read. As one of our interviewee noted, "reading a source code written by someone you do not know requires a lot of skill." When OSSD becomes more widely spread, teaching programmers how to read, not only how to write, will become a necessity not a luxury.

Incentives for open source stakeholders. Common to all the three types of OSSD, it seems that much more human and social factors are involved than conventional software development: a sense of pride, feeling of participation, and leadership of a community. We need a better understanding of how virtual organization and community evolves, and how people formulate a community by looking at other disciplines such as sociology, anthropology, and economics. For instance, Gallivan studies the role that trust plays in OSSD [5]. This type of study needs to be better integrated with software engineering framework supporting OSSD.

6. ACKNOWLEDGEMENTS

This research has been partially supported by Information-processing Promotion Agency (IPA), Japan. We thank the project leaders and developers who participated in our case study. Finally, we thank Jonathan Ostwald for his profound comments and suggestions on earlier versions of this paper.

7. REFERENCES

[1] A. Aoki, K. Hayashi, K. Kishida, K. Nakakoji, Y. Nisinaka, B. Reeves, A. Takashima, Y. Yamamoto, A Case Study of the Evolution of Jun: an Object-Oriented Open-Source 3D Multimedia Library, Proceedings of International Conference on Software Engineering, Toronto, CA., IEEE Computer Society, Los Alamos, CA., pp.524-533, May, 2001.

[2] BSD license, http://www.opensource.org/licenses/bsd-license.html

[3] Fielding, R.T. Shared Leadership in the Apache Project, Communications of the ACM, Vol.42, No.4, ACM, New York, NY, pp. 42-43, April, 1999.

[4] Free Software Foundation, http://www.gnu.org/fsf/fsf.html

[5] Gallivan, M.J., Striking a Balance between Trust and Control in a Virtual Organization: A Content Analysis of Open Source Software Case Studies, Information Systems Journal. Blackwell Science, (submitted), 2001.

[6] GNU General Public License, Free Software Foundation, http://www.fsf.org/copyleft/gpl.html.

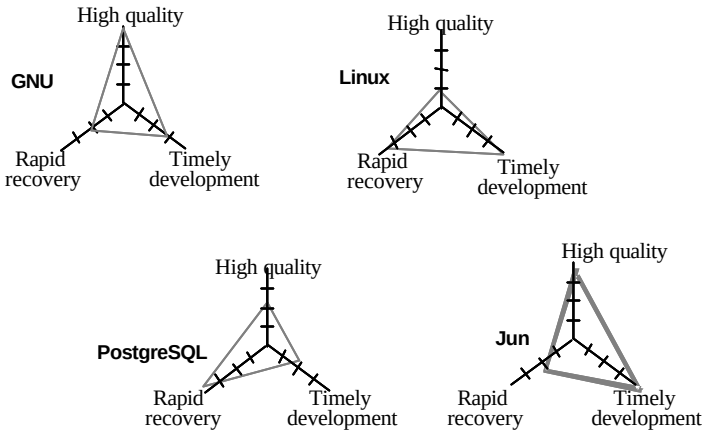

Figure 4;: Characterizing the Four Projects

[7] GNU Software, http://www.gnu.org/

[8] GNU Philosophy, http://www.gnu.org/philosophy/philosophy.html

[9] GNU Coding Standards, http://www.gnu.org/prep/standards_toc.html

[10] GNU Copyleft, http://www.gnu.org/copyleft/copyleft.html

[11] Godfrey, M.W., Tu, Q., Evolution in Open Source Software: A Case Study, Proceedings of the 2000 International Conference on Software Maintenance, San Jose, California, October 2000.

[12] Jun, http://www.sra.co.jp/people/aoki/jun/

[13] Koch, S., Schneider, G., Results from Software Engineering Research into Open Source Development Projects Using Public Data, 2000 http://opensource.mit.edu/papers/koch-ossoftwareengineering.pdf.

[14] Mockus, A., Fielding, R.T. & Herbsleb, J., A Case Study of Open Source Software Development: The Apache Server, Proceedings of International Conference on Software Engineering, Limerick, Ireland, ACM Press, pp. 263-272, June, 2000.

[15] Moon, J.Y. & Sproull, L., Essensce of Distributed Work: The Case of the Linux Kernel, First Monday: Peer-reviewed Journal on the Internet, 2000, http://www.firstmonday.org/issues/issue5_11/moon/index.html.

[16] Open Source Initiative (OSI), http://www.opensource.org/

[17] O'Reilly, T. Lessons from Open-Source Software Development. Communications of the ACM, Vol.42, No.4, ACM, New York, NY., pp. 33-37, April, 1999.

[18] OSI Business Support, http://www.opensource.org/advocacy/case_for_business.html

[19] Ousterhout, J., Free Software Needs Proft, Communications of the ACM, Vol.42, No.4, ACM, New York, NY., pp. 44-45, . April, 1999.

[20] PostgreSQL, http://www.PostgreSQL.org/

[21] Raymond, E. The Cathedral and the Bazaar, http://www.ccil/org/~esr/writings.

[22] Torvalds, L. The Linux Edge, Communications of the ACM, Vol.42, No.4, ACM, New York, NY., pp. 38-39, April, 1999.

[23] Wall, L. The Origin of the Camel Lot in the Breaskdwon of Bilingual Unix, Communications of the ACM, Vol.42, No.4, ACM, New York, NY., pp. 40-41, April, 1999.