Code contribution practice in Linux kernel: How product structure shapes organization

Ben Trovato[*]
Institute for Clarity in Documentation
1932 Wallamaloo Lane
Wallamaloo, New Zealand
trovato@corporation.com

G.K.M. Tobin[†]
Institute for Clarity in Documentation
P.O. Box 1212
Dublin, Ohio 43017-6221
webmaster@marysville-ohio.com

Lars Thørväld[‡]
The Thørväld Group
1 Thørväld Circle
Hekla, Iceland
larst@affiliation.org

Lawrence P. Leipuner
Brookhaven Laboratories
Brookhaven National Lab
P.O. Box 5000
lleipuner@researchlabs.org

ABSTRACT

We investigate how the organization of contribution team and culture are affected by product structure (legacy product) in Linux kernel. In FLOSS projects, code commit privilege is often employed to ensure code quality. Code committers are gatekeepers of code repository, and responsible for committing code for contributors who author the code but don't have the privilege to commit. The number of committers compared to the number of authors represents how the contribution team is organized and suggests a congruency of work load and communication in the team.

Using code change history and developer interviews we describe the evolution of contribution practice in the diverse circumstances and find that a) The product structure involves not just modules and cross-cutting concerns, but also information retrieval strategies and other activity structure; b) Product structure has a dramatic effect on the organization of contribution team (learning reproduces organization through product structure.). In particular, the loose coupled (well modularized) modules like drivers would have team fluctuating at size 20 (one committer works for ten authors), and close coupled modules like kernel would have a size of 10. c) The product structure is shaped by business requirements and active contribution inflow.

We expect our findings could be used to improve FLOSS project contribution process by describing ways of organizing contribution teams according to product structure characterized by features and contribution activities. The findings also suggest that software teams maintaining legacy modules are likely to maintain the original (initial) culture and may be able to adjust to changing environment but with a delay.

Keywords

product structure, code commitment organization

1. INTRODUCTION

Linux kernel has been a legend in the Free/Libre and Open Source Software (FLOSS) world. Linux kernel based distributions (operating systems) established a commercial success that many other FLOSS projects would like to pursue. For example, they take 98.8% positions in the top 500 fastest supercomputers in Nov 2015[1]. The statistics from monitoring a substantial number of web sites during the last twelve months (until Dec 2015) show Linux kernel based web clients have a 28.89% share in the market.

The success of Linux kernel can not be achieved without the contributions from participants. Starting from Linus Torvards, volunteers played a crucial role in the development of linux kernel. However, the number of volunteers (unpaid developers) contributing to the Linux kernel has been slowly declining for many years, now sitting at just 12.4% (it was 13.6% in 2014, and 14.6% in 2013). Meanwhile, commercial participation substantially grows in recent years, large companies like RedHat and Intel put substantial resources on the development of Linux kernel, e.g., Intel contributed 10.5% of changes, ReHat 8.4% in 2014[2]. Now more than ever, the development of the Linux kernel is a matter for the professionals, as unpaid volunteer contributions to the project reached their lowest recorded levels in the latest ”Who Writes Linux”

[*] Dr. Trovato insisted his name be first.

[†] The secretary disavows any knowledge of this author's actions.

[‡] This author is the one who did all the really hard work.

© 2016 ACM. ISBN 978-1-4503-2138-9.
DOI: 10.1145/1235

[1] https://en.wikipedia.org/wiki/Usage_share_of_operating_systems

[2] https://s3.amazonaws.com/storage.pardot.com/6342/120970/lf_pub_who

report[3].

As for why Linux is now mostly developed by well-paid engineers, the possible reasons are myriad. The most obvious and compelling reason is that these big companies have a commercial interest in the continued good health of Linux. 10 years ago, Linux was the plaything of hobbyists and supercomputer makers – today, it powers everything from smartphones (Android) to wireless routers to set-top boxes. The continuing commercial interest in Linux is highlighted by another statistic from The Linux Foundation report: In mid-2011, only 191 companies were involved in the Linux kernel; by the end of 2013, that number was up to 243[4].

Apparently, Linux kernel experienced dramatic change since the very beginning, e.g., substantial expanding of kernel code and increasing of commercial participation. In particular, different modules of Linux kernel present different nature and attract different contributors. For example, the module of drivers accounts for the largest proportion of linecount (56.6%) in the kernel, probably because various of hardware manufacturers have been trying to get their drivers into the kernel and devoted substantial effort in their cause. As for the module of kernel, the very central one, attracting the most capable and ambitious developers in the world, takes only 1.2%. Therefore, it is of interest to understand if the contribution practice of different modules of Linux kernel differs from each other, and how they evolve over time adapting to different business environments. In particular, we aim to answer the following questions:

- Do different modules of Linux kernel have different contribution practice from each other?
- Do different modules of Linux kernel evolve their practice over time and how?

On the one hand, it may help us understand the new challenges in the new FLOSS landscape like commercial participation. On the other hand, the understanding can help us utilize the best practices and amplify the effect.

We retrieve the code commits from the mainline repository of linux kernel, and use the data to quantify the community contribution practice. We focus on one particular factor that tries to measure the team relationship between code authors and code committers representing contribution organization. We found the module of drivers is unique among all the modules in terms of having the biggest team size and organized more spontaneously. We found the ratio of number of authors over number of committers is decreasing over time for all the modules. Even both the number of authors and committers are increasing (the module of drivers is the single module that has a sharp increase for both that is different from the other modules), apparently, the increase of committers is faster than that of authors. That may suggest a more professional organization of the working teams has been happening in the community. Moreover, the ratio of each module correlates with the number of new joiners, and the number of new LTCs. This may imply that a looser control of working team brings more outsiders which requires attention from the community.

The rest of the paper is organized as follows. Section 2 discusses the related work. Section 3 describes the research methodology used in our study. Section 5 presents the results of our study. Section 7 discuses limitations of our study. We discuss practical implications of our empirical results in Section 6 and conclude in Section 8.

2. BACKGROUND AND RELATED WORK

The nature and performance of FLOSS development are subject of numerous investigations, but studies on evolution of contribution practice along the change of project landscape particularly commercial participation are far less common.

An early study of Apache web server and Mozilla web browser [3] quantified various aspects of OSS development practices. The results were framed as seven hypotheses that outline key aspects of OSS development. In this paper we focus on a different aspect of contribution organization.

Community strategies and practices are often addressed in the literature, e.g., community architecture [3, 6], license and intelligence property management mechanism [**?**], and code commit privilege or ownership control [3, 6, 5], etc. Meneely and Williams [2] examined the relationship of the number of developers working on parts of the Linux kernel with security vulnerabilities. They found that when more than nine developers contribute to a source file, it is sixteen times more likely to include a security vulnerability.

Unlike in prior work, we observe the contribution practice presented by Linux kernel experiencing various of technical and economic landscape and quantify the important aspects of the development that are likely to be affected by the product structure: contribution organization.

3. METHODOLOGY

3.1 Data preparation

We retrieved all the commits from the mainline repository of Linux kernel[5]. We took steps to clean and standardize the data, and obtained a data level for the further analysis. Table 1 shows an observation used for this study. Each observation is a change committed to the mainline repository maintained by Linus Torvalds. It records who and when writes the code, and who and when commits the code, the author and the committer may be the same person, but most of the time (74%) they are not. The changed file is represented by its directory in the code repository. For example, "drivers/pci/iova.c" illustrates that the changed file is under the directory of pci driver.

We followed the following rules to obtain the final changes for this study. 1), we only look at c files. 2), we consider the first level modules retrieved from the file path as the module structure used in linux kernel. Overall we obtained 22 modules. The top seven modules include drivers, arch, fs, net, mm, kernel, sound. 3), We consider a developer's joining time as the time they author their first commit.

4. PRODUCT STRUCTURE AND METRICS

4.1 Product Structure

[3] linux.slashdot.org/story/15/02/18/1745246/torvalds-people-who-start-writing-kernel-code-get-hired-really-quickly

[4] www.extremetech.com/computing/175919-who-actually-develops-linux-the-answer-might-surprise-you

[5] git://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git

Table 1: Attributes of an observation

author	author time	committer	commit time	changed file	module
Minfei Huang	Nov 6 16:32:45 2015 -0800	Linus Torvalds	Nov 6 17:50:42 2015 -0800	kernel/kexec.c	kernel

We have observed two aspects of product structure: the architecture, which includes several structures, of which we will primarily focus on the module structure, and the development activity structure.

Based on our interviews and prior experience, developers' tasks are assigned based on these two types of structure. In our study the module structure was organized according to product package/subsystem and functionality (functionality, such as internationalization, may cut across the package/subsystem boundaries).

The activity structure followed common development practices, such as building, installing, configuring, and testing the product. It also included practices used to fix and report problems and to design and develop new features. Furthermore, underlying these generic practices, there were substantial differences in information seeking behavior needed to accomplish these common tasks, for example, knowing when and how to inspect the execution log or where to find information about similar bugs that occurred in the past, and variation in acceptable norms, such as how many defects are acceptable, and what should be tested, how it should be tested, and how extensively it should be tested.

Based on our observations, the way each practice was implemented was carried over from the original practice used to develop the products, often with no individuals serving as conduits. For example, when fixing defects Project A extensively used their rich problem resolution repositories, while project C used almost exclusively the logs of product execution and project B focused on latest code changes. In fact, given the nature of the products such strategies make considerable sense. Project A was very difficult to install and run and did not have well defined states, making execution logs less valuable for debugging. On the other hand, Product C could be intuitively thought of as a state machine with well defined states and transitions, suggesting that execution logs represented a nearly optimal way to understand the nature of a problem. Therefore, we hypothesize that the activity structure is in fact a part of the product structure that is either enforced by the particular product domain and its architecture, or encoded in the historic information repositories (code, execution traces, and tracking systems) of how the product was constructed and maintained in the past.

Our claim is not that the mere fact that the original team and the new team perform testing implies some learning from the product structure (we expect that most software developers know about testing from their undergraduate studies). Rather, the similarity between the ways testing was done in the original and the new team indicates that the product itself has some effect on learning and that product structure incorporates development activity structure. We propose that developers learn through performing regular project tasks under the constraints ("guidance") of the product structure, and, accordingly, change their positions in the project/organization. The centrality of a task, embodies the centrality of the modules or activities the task is related to, and reflects the centrality of the position the developer has in the organizational communication.

4.2 Metrics

Code ownership.

Ownership is a key aspect of large-scale software development, a valid proxy for expertise. Strong ownership, i.e., a single key developer responsible for a particular component in a system (whether it be a file, class, module, plugin, or subsystem), might be more effective in carrying out all tasks consistently and to completion [4]. Low communication overhead is required. External communications occur through a single channel. However, the disadvantages are obvious. It might be necessary to trade overall development time for development efficiency. The system will have a low truck number – only one developer need be hit by a truck to kill the project. Rogue individuals might "own" their code to the exclusion of organizational goals. It's common for commercial organizations to enforce strong code ownership in order to achieve good quality, see, e.g., [1]. Given the informal, distributed way in which FLOSS projects like linux kernel are built, it seems that rather than any single individual writing all the code for a given module, those in the core group have a sufficient level of mutual trust that they contribute code to various modules as needed [3]. In FLOSS projects, code "ownership" to be more a matter of recognition of expertise than one of strictly enforced ability to make commits to partitions of the code base.

One measure of ownership is how much of the development activity for a component comes from one developer. If one developer makes 80% of the changes to a component, then we say that the component has high ownership.

The proportion of ownership (or simply ownership) of a contributor for a particular component is the ratio of number of commits that the contributor has made relative to the total number of commits for that component. Thus, if Cindy has made 20 commits to ie9.dll and there are a total of 100 commits to ie9.dll then Cindy has an ownership of 20%.

Ratio of authors to committers.

We focus on one particular factor that tries to measure the team relationship between code authors and code committers. Code authors in this study are people who write the code that gets into the mainline repository. Code committers are people who have the privilege to write the repository. A committer may be responsible for a specific module (functionality), so responsible for committing whoever's code on that module.

Code authors in this study are people who write the code that gets into the mainline repository. Code committers are people who have the privilege to write the repository. A committer may be responsible for a specific module (functionality), so responsible for committing whoever's code on that module. Naturally, the ratio of number of authors over number of committers may represent the load of a commit-

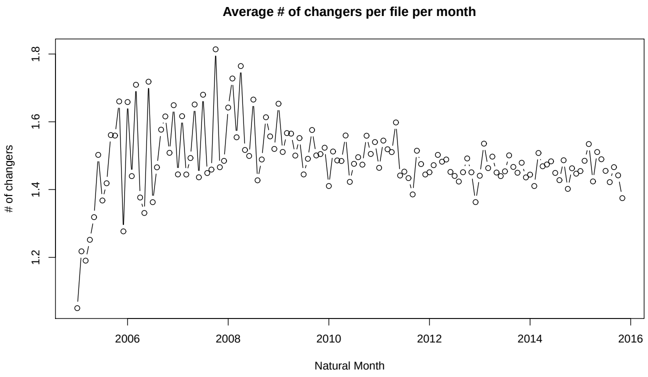

Figure 1: Code ownership

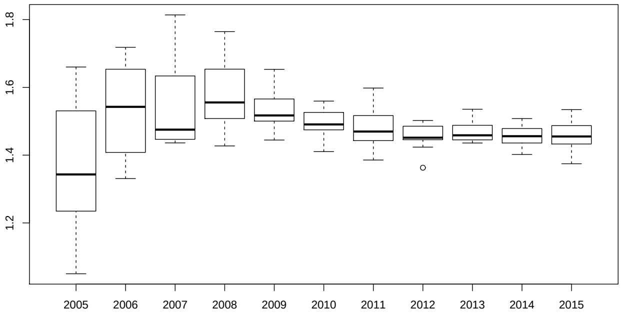

Figure 2: Boxplot of code ownership

ter, or the difficulty (easiness) of the module, or, how hard it is for the author to get her code in. And the change of the ratio on the same module may represent the change of project landscape (e.g., new features may attract more authors but committers may keep at the same level as before), or the maturity of the module (e.g., a mature module may reduce committers).

5. RESULTS

5.1 Code Ownership

Commercial involvement may bring an organized structure to the development process in addition to dedicating employees' time to find and report issues. In particular, companies might encourage stronger code ownership in contrast to loose code ownership observed in Apache and Mozilla projects [3].

Stricter code ownership would manifest itself as a smaller number of developers changing a single file. The average number of changers per file ($Changer_{avg}$) may, therefore, be used to measure the strictness of code ownership. We calculated $Changer_{avg}$ for each month and the resulting time series of this measure of code ownership are shown in Figure 1. We can see a pronounced increase of this measure from 2005 to 2008, then a decrease until today. The boxplot in Figure 2 (every boxplot has 12 numbers for every year) shows a clearer trend[6].

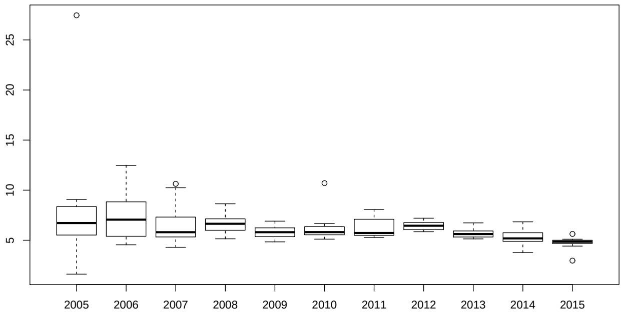

Figure 3: Boxplot of developer productivity

The kernel which forms the core of the Linux system is the result of one of the largest cooperative software projects ever attempted [?]. The stricter code ownership may be developed to simplify the cooperation in the building of more and more complicated kernel.

At the same time, we found the average number of different files touched by a developer per month decreases over time as shown in Figure 3, this may be due to the fact that the code is becoming more and more complicated, and therefore tasks are becoming more and more difficult. It may require more effort to work on multiple files, and the interconnections between files may be reduced in order to reduce the needs of cooperations.

Moreover, the average number of modules a developer touches decrease over time, together with the average number of changes committed by a developer (author).

In summary, we have the following observation:

Observation 1. The kernel developers have reduced their scope of activities (number of different files they touched per month), but their code ownership is intensified (number of people who touched a file per month).

This may suggest that they put more effort on single files instead of working with others on the same files. This indicates a possible phenomenon that requires attention from the kernel community: if people start to work on their own, there is a risk that the cooperation critical for the health of the community is reduced.

5.2 Organization of Code Contribution

we focus on one factor that tries to measure the team relationship between code authors and code committers. Code authors in this study are people who write the code that gets into the mainline repository. Code committers are people who have the privilege to write the repository. A committer may be responsible for a specific module (functionality), so responsible for committing whoever's code on that module. Naturally, the ratio of number of authors over number of

[6] The lower and upper boundaries of the rectangle of the boxplot represent the first and the third quartile and the thick line is the median. The lowest and upper horizontal lines represent the 1.5 quartiles away from the mean. The points below or above the horizontal lines are suspected outliers.

committers may represent the load of a committer, or the difficulty (easiness) of the module, or, how hard it is for the author to get her code in. And the change of the ratio on the same module may represent the change of project landscape (e.g., new features may attract more authors but committers may keep at the same level as before), or the maturity of the module (e.g., a mature module may reduce committers).

We use this measure to investigate how the code contribution is organized and how the balance between authors and committers is achieved in the community. We take three years as a window, start from January 2005 (the window is from January 2005 to December 2007) and roll the window by month until the window getting to December 2015, i.e., the date we retrieved the data. We calculate the metrics on these windows. Considering the variations of the modules, we look at the team organization on each module separately. Figure 4 presents the ratio changes in a three-year window rolling from January 2005 month by month.

Overall, the ratio is decreasing over the years. Even both the number of authors and committers are increasing (drivers is the single module that has the sharpest increase for both), apparently, the increase of committers is faster than that of authors. That may suggest a more professional organization of the working teams has been happening in the community.

As we can see, the module of drivers (the top line) has a much higher ratio than the other modules. It fluctuates at around 20 compared to 10 that the other modules move around. If we look at the number of authors and committers separately on the modules, we can see that they both grow over time on all the modules, but the drivers module has the biggest growth. At the same time, even both authors and committers grow, apparently committers have a slower growth, therefore the ratio drops.

We discovered that the ratio is around 10 for most modules but high above 10 for the module of drivers. The ratio for modules like kernel, mm stabilizes at 6 or 7, a reasonable size that is a controllable number for a team. A further investigation reveals that the drivers module have a much looser control over the team, the fact that a committer works for 20 authors suggests that the tasks may be easy on this feature. Net drivers and staging drivers appear to be on the easiest tasks, suggesting a place where newbies like to start. The fact that the drivers module has more than 50% share of newcomers supports this hypothesis (what's the proportion of changes of drivers?). In fact more than 40% of all the committers in the kernel community started from the drivers module (no comitters of kernel module or mm module started from kernel or mm).

We take a further investigation to understand the story in the module of drivers. Figure 5 shows the ratios of subdirectories under the drivers. We can see net, staging and media are the three sub-modules that are different from the others that have the similar trend compared to the modules at the first level, i.e., they fluctuate at 10.

The investigation on net module shows 10 is a common number for this metric, as presented in Figure 6. Actually except mac80211, all the other sub-modules fluctuates at 10 (sunrpc seats at five). A closer look at mac80211 clarifies its change: the number of committers is from 3 to 8 and back to 3, but the number of authors stays the same, making the curve change sharply.

In summary, 10 might be a reasonable number for organizing the contribution team. If the code is well modularized,

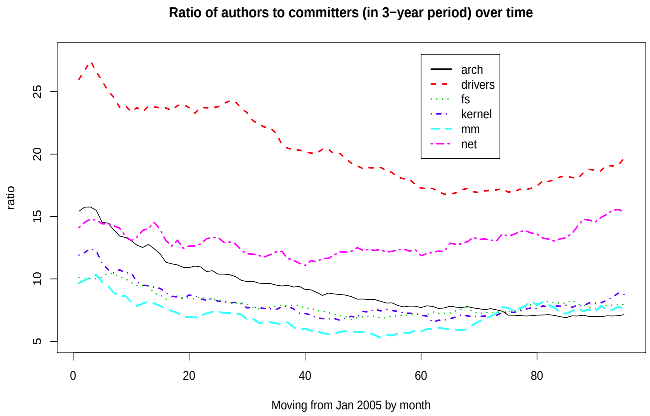

Figure 4: Ratio of #authors over #committers

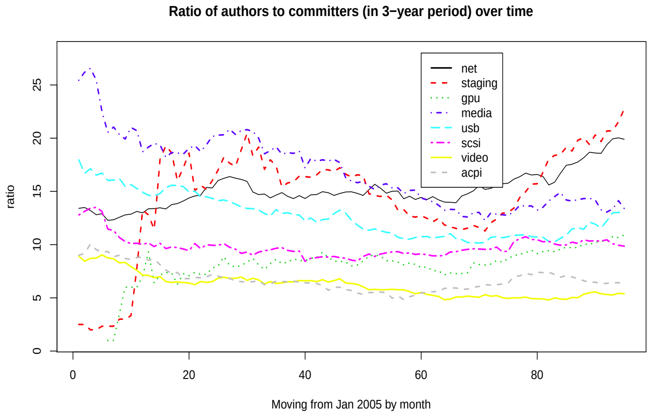

Figure 5: Ratio of #authors over #committers on Drivers

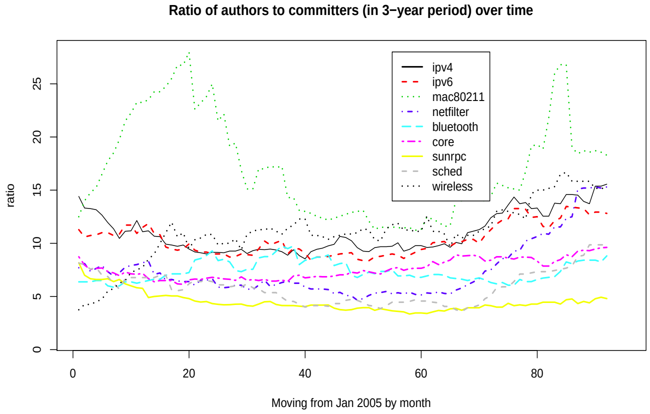

Figure 6: Ratio of #authors over #committers on Net

the ratio could be bigger, otherwise it could be smaller.

Through chcking with the core members of kernel, we found the ratio changes because of the following reasons:
1), when there is new feature, a bulk of authors would cause fluctuation of this measure, because the committers wouldn't increase at the moment, e.g., drivers/staging/iio.
2), when there is new feature, the increase of committers is likely to be behind the increase of authors, e.g., net/wireless, drivers/staging/comedi, drivers/staging/iio.
3), for the matured feature, it's likely the committer would leave, and a small amount of leaving committers would lead to a drama increase of this ratio because the proportion is relatively high, e.g. net/mac80211.
4), in general the development is stable.

In summary, we have the following observation:

Observation 2. *The suitable way of organizing code contribution team in Linux kernel is to have a committer work for ten authors. If the code is well modularized, the ratio could be bigger, otherwise it could be smaller.*

The ratio for modules like kernel, mm stabilizes at 6 or 7, a reasonable size that is a controllable number for a team. A further investigation reveals that the drivers module have a much looser control over the team, the fact that a committer works for 20 authors suggests that the tasks may be easy on this feature. Net drivers and staging drivers appear to be on the easiest tasks, suggesting a place where newbies like to start. The fact that the drivers module has more than 50% share of newcomers supports this hypothesis (what's the proportion of changes of drivers?). In fact more than 40% of all the committers in the kernel community started from the drivers module (no comitters of kernel module or mm module started from kernel or mm).

Moreover, the ratio of each module correlates with the number of new joiners, and the number of new LTCs. This may imply that a looser control of working team brings more outsiders.

6. DISCUSSION

7. LIMITATION

8. CONCLUSION

9. REFERENCES

[1] C. Bird, N. Nagappan, B. Murphy, H. Gall, and P. Devanbu. Don't touch my code!: Examining the effects of ownership on software quality. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE '11, pages 4–14, New York, NY, USA, 2011. ACM.

[2] A. Meneely and L. A. Williams. Secure open source collaboration: an empirical study of linus' law. In *Proceedings of the ACM 2009 Conference on Computer and Communications Security.*

[3] A. Mockus, R. T. Fielding, and J. Herbsleb. Two case studies of open source software development: Apache and Mozilla. *ACM Transactions on Software Engineering and Methodology*, 11(3):1–38, July 2002.

[4] I. Nordberg, M.E. Managing code ownership. *Software, IEEE*, 20(2):26–33, Mar 2003.

[5] G. von Krogh, S. Spaeth, and K. R. Lakhani. Community, joining, and specialization in open source software innovation: a case study. *Research Policy*, 32(7):1217–1241, July 2003.

[6] Y. Ye and K. Kishida. Toward an understanding of the motivation of open source software developers. In *ICSE 2003*, pages 419–429, Portland, Oregon, 2003.