ABSTRACT
As software projects evolve over time, source code inherently becomes more complicated and tend to drift away from its original structure envisaged by the project founder. Uncontrolled software complexity makes it difficult to comprehend, modify code, and maintain sustainable level of OSS developers. Faced with challenges of increasingly complicated software design, software refactoring can be one solution to improve software design and quality. However, it requires substantial coordination among developers involved. Especially, it is challenging for community-based OSS projects in which there is no manager who tells what to do or resources to pay developers. This study qualitatively and quantitatively explores how refactoring that touches source code across different parts of software application—large-scale refactoring—was made to improve software quality. We conducted a case study of GNU grep, a widely used software program. We sought to solve a puzzle of how it was carried out despite lack of capabilities to overcome coordination challenges, based on the analysis of three episodes, two unsuccessful and one successful. In this study, we highlight the organizational circumstances for large-scale refactoring in successful, community-based OSS projects. This study provides concrete, actionable insights about OSS practices, the benefits and challenges of refactoring in community-based OSS projects.

Keywords
Refactoring, Open Collaboration, Open source software, Coordination

1. INTRODUCTION
Open source software (OSS) projects change over time. The code inherently become more complicated and tend to drift away from its original structure envisaged by the project founder, which risks of lowering software quality. Further, if OSS complexity is uncontrolled, it is likely to make it difficult to comprehend, modify code, and maintain sustainable levels of OSS developers who can write code, considering the super-linear growth of OSS over time [12, 34].

Accordingly, refactoring open source code can be one solution to make it easier to comprehend code and make developers more productive. Software refactoring is considered a key activity in the software development process. It aims to improve software quality such as complexity, maintainability, performance, and readability [28]. Fowler [11, p.46] defines refactoring “a change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behavior”. That is to say, refactoring makes architectural changes while preserving the behavior of the codebase.

Software refactoring has been intensively studied in the context of commercial software [19;21;28], and a hybrid-mode of open source products [20;33;37]. In practice, software engineers at Microsoft perceive refactoring codes involves substantial efforts, costs, potential risks, and needs the support from a variety type of tools [21].

While software developers in commercial organizations are bound to work on the assigned tasks, community-based OSS projects lack capabilities to overcome substantial coordination. Further, voluntary OSS contributors are independently motivated. In this aspect, the finding reported from the survey of commercial organization [21] suggests that refactoring is likely harder in the context of community-based OSS projects. In addition, although refactoring can be one solution to improve OSS quality, relatively little is known about how refactoring work could be carried out in community-based OSS projects in which there is no manager who tells what should be done or resource to pay developers who work on refactoring.

In this study, we explored organizational circumstances to refactoring code across different parts of software application (i.e., large-scale refactoring), and the impact of large-scale refactoring on both technical and organizational dimensions by investigating three episodes from the GNU grep project. To explore the impact of large-scale refactoring on both technical and organizational dimensions, we examined the differences in software metrics including McCabe’s cyclomatic complexity, which is used as a proxy to measure the cognitive burden on a software developer in comprehending source code, between two successive versions of grep 2.5.4 and 2.6, as well as developer base over time.

Our finding reveals that large-scale refactoring that touches the code across different parts of grep were carried out by two newly assigned maintainers and they shifted towards traditional coordination mechanisms: explicit division of labor, firm co-membership outside the OSS project, and direct, interpersonal communication outside the project. In contrast, two earlier attempts to restructure grep code remained uncommitted due to the constraint of resource. Our finding of refactoring benefit is consistent with a software engineering view, in that large-scale refactoring strikingly reduced structural complexity and the amount of code. Our finding also suggests that it appears to be
related to an increasing trend in grep developers who can write the code after refactoring.

The remainder of this paper is structured as follows: section 2 provides theoretical background about the benefits of software refactoring and the challenges of software refactoring, especially regarding coordination of refactoring activities. Section 3 introduces research setting, GNU grep, and methodology to identify refactoring episodes and commits, and to measure software metrics. Then, we illustrate three episodes, and argue how large-scale refactoring could be performed despite substantial refactoring effort and coordination. Then, we discuss our exploratory quantitative analysis of software metrics in two successive versions of GNU grep and developer base over time. We conclude and highlight the implications of our study for both researchers and practitioners.

2. Theoretical background

2.1 The Benefits of Software Refactoring

Software refactoring refers to transformations that maintain the behavior of software while improving aspects such as maintainability, performance, and readability [28]. Refactoring is needed to convert legacy or obsolete code into a more modular or structured form, or even to migrate code to a different programming language or language paradigm, for instance from C language to C++ language [9]. Fowler (1999) [11] links bad smells of codes to refactoring and those codes have symptoms such as duplicated code, long method, and long parameter list. He also catalogues refactorings such as Extract Method, Extract Interface, and Rename Method with a proper IDE to remove those bad smells.

In a technical dimension, the potential benefits of software refactoring include reduced software complexity, increased comprehensibility of source codes, which makes it relatively easier to maintain software. For instance, the purposeful redesign effort is found to make Mozilla more modular [26]. Mozilla’s core developers, who mostly worked for Netscape, purposefully redesigned Mozilla before releasing it under open source license. The redesign of Mozilla aimed to remove redundant or deteriorated source files, to rewrite major parts of the code to improve performance, and to restructure the product by grouping files into smaller modules. The redesign of Mozilla resulted in an architecture that was significantly more modular than that of its predecessor. In contrast, while it has been widely believed that refactoring improves software quality, a study of refactoring a large system, Microsoft Windows 7 found that refactoring benefit was not consistent across various software metrics [21]. The empirical investigation of software metrics of popular OSS projects found that refactoring does not seem to improve software quality and in some cases, software metrics worsen after refactoring. The authors in that study [37] explain that it can be because either refactoring may not improve software quality in a measurable way or refactoring was not effective enough to improve software quality.

In an organizational dimension, the purposeful redesign effort before releasing Mozilla under open source license aimed to become more attractive potential contributors [26] because modular architecture enables OSS contributors to understand code without having to learn whole systems and affect other’s tasks. The study on structural complexity of OSS projects found that the structural complexity negatively affected the number of contributions from new developers [29].

In brief, there have been intensive studies on refactoring benefits in the codebase, however, we have confusing evidence. Hence refactoring benefits continuously call into the question about whether refactoring improves software quality. In addition to the impact of refactoring on the codebase, this study examines another dimension of refactoring benefit: In the context of community-based OSS projects, it is important to investigate the impact of refactoring on organizational dimension such as maintaining sustainable level of developer base. This is because, if refactoring improves software quality such as program readability, complexity, then, it should be more attractive to OSS contributors. Accordingly, we investigate the impact of refactoring open source code on technical as well as organizational dimensions.

2.2 The Challenges of Software Refactoring: Coordination of refactoring activities

According to Mens and Tourwe (2004) [28], refactoring activities comprise a number of distinct activities: 1) Identify where the software should be refactored. 2) Determine which refactoring(s) should be applied to the identified places, 3) Guarantee that the applied refactoring preserves behavior. 4) Apply the refactoring. 5) Assess the effect of the refactoring on quality characteristics of the software, and 6) Maintain the consistency between the refactored program code and other software metrics. Beck and Fowler (1999) [11] highlights that it is important the whole team should become aware of that big refactoring (e.g., converting procedural design to objects and extracting hierarchy) is “in play” (p.293) and make their moves accordingly. This process may require to stop progress for a couple of months while large-scale refactoring is performed. Accordingly, each activity to carry out refactoring likely requires intense coordination among developers or teams, particularly if refactoring should be carried out across different parts of software system. Organizational aspects of refactoring activities are reviewed below.

A survey of 328 professional software engineers at Microsoft reveals the inherent challenges related to refactoring practice [21]. The survey result reported that a large number of the respondents (71%) considered that refactorings catalogued by Fowler [11] often part of large, high-level effort to improve software. 28% of respondents reported the challenges such as working on large code base, generating the need for coordination with other developers and teams, and ensuring program correctness after refactoring [21]. More importantly, their interviews with six key members of a designated refactoring team within Microsoft found that the team made decisions on refactoring after substantial analysis of software structure and the system-wide refactoring effort was centralized and top-down [21].

While commercial software organizations are characterized by co-membership, explicit and shared goals and authority-based planning or oversight, community-based OSS projects are characterized by fluid, porous membership, independently motivated participation, and informal relationships [25]. Accordingly, the challenges of refactoring reported from the survey of commercial organization [21] suggests that working on refactoring is likely harder in the context of OSS production communities because open development cannot be as tightly planned and managed as most commercial software.
The challenges of refactoring in community-based OSS projects were also studied through participation observation of a project called “Bibdesk” [17]. Bibdesk was, and still is, a community supported project with no formal institutional existence and no significant corporate participation. The authors describe a period in the project called “Bibdesk 2.0,” which was an effort to release a refactored and largely rewritten version of Bibdesk. However, BibDesk 2.0 was never completed, despite agreement among the project members on its need. Howison and Crowston [17] explain that the Bibdesk 2.0 episode was problematic because such work deviated from the normal pattern of independently motivated, incremental small steps by individual developers (which they term “open superposition”). While only a single developer sought to undertake the overall refactoring, other developers were asked to freeze their work during the period. As the refactoring period became longer and longer, other developers returned to adding contributions to the un-refactored codebase, eventually rendering the refactored branch out of date, increasing the amount of work to be done on the refactored branch, making it longer and thus driving a cycle that lead to the refactoring effort being abandoned.

A case study of GRASS (Geographic Resources Analysis Support System), a large open-source geographical information system, also describes the challenges of refactoring such a large system and ascribes failure to a lack of resources [8].

The investigation of large-scale refactoring the Eclipse Help subsystem reveals potential risks of big refactoring in the context of OSS production [30]. Five skilled Eclipse engineers contributed in planning and working on the restructuring tasks, which aimed to provide a minimal platform, called Rich Client Platform (RCP). Those engineers worked for IBM and had both organizational support and tool support. However, the main obstacle was found that they had lack of shared understanding in the codebase, and the coordination of interdependent and detailed changes in the codebase.

Another case study of Apache Jackrabbit [36] suggests that the coordination of software development projects is affected by refactoring code. Particularly, the authors in that study found that core developers are found to send more messages while refactoring code than in previous, non-refactoring, periods. That is to say, it shows that refactorings require additional coordination efforts among developers who are involved in, and they shifted towards a traditional coordination mechanism such as direct communication.

Taken together, then, prior studies establish that large-scale refactoring is difficult but worthwhile. Studies argue that refactoring can be valuable for open source projects, establishing a basis for improved software design, and attracting new contributors (Mozilla). Yet prior studies also establish that refactoring is difficult for open source projects, both because it requires greater effort that might not be available (GRASS), and intensive coordination between developers (Eclipse Help subsystem, Apache Jackrabbit) and because it delays other work, simultaneously undermining commitment to the refactoring effort and making the refactoring task more difficult as new features or bug-fixes “pile up” and have to be worked into the refactoring after it is begun (Bibdesk).

Thus it is potentially very valuable to understand how to overcome the challenges of refactoring in the open source environment, and particularly to consider the organizational requirements, too often unexplored in the literature on refactoring.

In this respect the prior literature is only of limited help. This is because most studies report on failed refactoring efforts, while those who have studied successful efforts (e.g., [26]) have concentrated on impacts and not processes. That is to say, in the context of OSS production, relatively little is known about how large-scale refactoring activities that require direct, intensive communication and coordination should be arranged. In this paper, we address this question directly, examining three episodes of attempted refactoring in a single project; the first two are unsuccessful but the third succeeds, allowing us to compare what worked to accomplish refactoring. Taken together, this study asks,

RQ 1: How could large-scale refactoring be undertaken despite lack of capabilities to overcome substantial coordination and large efforts?

RQ 2: What is the impact of large-scale refactoring on the codebase as well as developer base over time?

3. Method

3.1 Research setting: GNU grep

This study aims to explore and understand the nature of refactoring practice in OSS production and its impact on both technical and organizational dimensions. The purposes of this study led to purposive sampling which is appropriate when seeking to discover relevant concepts.

We examined a set of projects before settling on the GNU Global Regular Expression Print (grep) project, which has been widely used and considered successful, and went through large-scale refactoring between 2.5.4 version and 2.6 version.

Grep is a command-line utility. It searches a file for regular expressions (patterns) and prints the lines containing matches to it. GNU grep is installed by default on almost every distribution of Linux, BSD, and is also available for Windows. GNU and the Free Software Foundation distribute grep as part of their suite of open source tools.

Hence, exploring the grep project provides useful insights about how grep contributors could carry out refactoring and the impact of refactoring on the codebase and developer base.

3.2 Data collection: Identifying Refactoring episodes and commits

To identify large-scale refactoring events, we used two methods. First, we looked for messages from the project mailing list. Second, we mined refactoring commits from GNU grep’s version history with keywords related to refactorings [31]. Further, we conducted semi-structured artifact-based interviews with two GNU grep developers to explore organizational circumstances and compare differences between unsuccessful one and successful one.

3.2.1 Operationalizing episodes

Our unit of analysis in this study is the episode, which is defined as “events, processes, and practices that occur over time and have a beginning and an end” [2, p.2]. In operationalizing episodes as a unit of analysis, we sought messages exchanged via bug-grep mailing list. Large-scale refactorings should be discussed with other participants because it affects other’s work in the codebase, unlike small refactoring such as minor cleanups [11].
GNU grep has two mailing lists—bug-grep and grep-commit. Bug-grep mailing list has been used to discuss most aspects of grep including development and improvement requests, reporting bugs, and the submission of the patches since November 2004. Grep-commit mailing list archives log messages associated with code commits, which grep’s git has also archived. Accordingly, bug-grep mailing list is the primary source to identify refactoring episodes.

In the process of identifying episodes from the mailing list, we used a list of top ten keywords reported by a survey of 328 professional software engineers whose check-in comments contained a keyword refactoring: refactor, clean-up, rewrite, restructure, redesign, move, extract, improve, split, reorganize, rename [21]. With those keywords, we first started browsing bug-grep mailing lists because if refactoring to make architectural change at high level, it should be discussed in public. We looked for a refactoring trigger, which is a message initiated a refactoring issue with the patch. If a refactoring trigger was captured, then all related messages were read through. By doing this, we identified three attempted refactoring episodes.

As the episodes were identified, within an episode we focused on what triggered refactoring issues and what was discussed to proceed refactoring in what ways. We also sought to understand the characteristics of how refactoring work was done in order to characterize refactoring processes within an episode. Once the factors were captured within an episode, then, the analysis moved on to across episodes to illuminate the factors that enabled or constrained further refactoring activities.

From this process, we identified two earlier attempts to suggest refactoring grep code in November 2004 and March 2005. Each was undertaken by different grep contributors but neither led to refactorings being integrated into the code base. Another attempt made between November 2009 to March 2010 was identified a successful episode that carried out large-scale refactoring, between grep 2.5.4 and grep 2.6.

3.2.2 Mining commit logs
The GNU grep project manages and archives source code on git, a distributed version control system (VCS). To identify refactoring commits from version histories, we first obtained all commits made on GNU grep’s git. As specific period of time when grep developers performed large-scale refactoring—between grep 2.5.4 and grep 2.6—is identified, based on analysis of bug-grep mailing list, we obtained commits logs made during those periods of time. As commit logs made between grep 2.5.4 and grep 2.6 were extracted, we looked for refactoring commits with ten keywords relevant refactor [21], which we stated in section 3.2.1.

While a total of 216 commits were made between grep 2.5.4 and grep 2.6, 93 commits were detected as commits relevant refactoring. Based on the analysis of those commits, we also found that a total of 3 grep developers—two maintainers for grep 2.6, and another contributor—made those refactoring commits. 42 commits were made by one grep 2.6 maintainer, 50 commits from another grep 2.6 co-maintainer, and one commit from grep contributor.

3.2.3 Semi-structured artifact-based interviews
Archival data such as mailing lists and commits may not provide the complete context of decisions and communications to researchers without deep knowledge of the project [39]. In addition, all communication among OSS participants are not archived by the system because they also use different types of medium such as a real-time text chat or sometimes in person meetings (e.g., OSS conferences) in addition to the project mailing lists to discuss about project in detail [15]. Accordingly, to mitigate those limitations of archival data, we also conducted semi-structured artifact-based interviews with two GNU grep developers to explore different organizational circumstances between unsuccessful one and successful one. The interviews focused on discovery of three dimensions of organizational circumstances: 1) firm co-membership outside the OSS projects, 2) physical and temporal co-location, and 3) rich, interpersonal communication (outside the project’s mailing list). We conducted one interview via Skype and another one via email.

3.3 Measurement in the codebase and developer base over time
To calculate software metrics of grep code, every official version released in public were downloaded. The version available in public includes: the initial version on git, which was committed on November 1998, and from grep 2.5.1 to grep 2.20.0 as of this writing.

3.3.1 Software complexity
Reducing software complexity is one of purposes to work on software refactoring [1;21;28]. Complexity is defined as a measure of the resources consumed by a system, while interacting with a piece of software to perform a given task [5]. If a computer is interacting, then complexity is defined by the time to be taken to execute a program. If a developer is interacting, then complexity is defined by the difficulty of performing tasks such as coding, debugging, and testing [18]. That is to say, from the software developer’s perspective, software complexity refers to software characteristics that make it difficult to understand and to modify [7].

McCabe’s cyclomatic complexity [27] has been widely used as a measure to assess software structural complexity. Cyclomatic complexity is also used to evaluate decision density, which refers to the relative amount of decision or control paths in the codebase [19]. Thus, researchers consider it as cognitive complexity and further, cyclomatic complexity is considered a key factor of maintenance performance due to its impact on the critical activity of understanding codes [4]. Following this line of reasoning, in this study cyclomatic complexity is used as a proxy to measure the success of a refactoring effort.

In measuring McCabe’s cyclomatic complexity, we used a package called Understand [35], commonly used in obtaining static code metrics. The tool we used in this study has also been employed by other researchers to obtain and confirm their software metrics [6;25].

3.3.2 Other metrics in the codebase and developer base
To identify the differences of software metrics before and after refactoring activities, we tracked the evolution of grep’s size—lines of code, the number of files the number of functions, inactive lines, and preprocessor lines [32].

We also tracked each version of grep to examine the impact of refactoring on an organizational aspect—the total number of grep developers who made commits on git and the total number of
commits per release, using an open source code analysis tool—GitStats\(^1\), which produces the statistics of git history.

4. A Puzzle of refactoring open source code

Section 4.1 describes two episodes of earlier two attempts to restructure grep code, which remained uncommitted. Section 4.2 illustrates another episode of heavy refactoring practice which was carried out between grep 2.5.4 and grep 2.6 and which was committed to the project. Then, section 4.3 weaves the findings across episodes together to identify the factors that constrained or facilitated refactoring work.

4.1 Earlier two attempts to refactoring

4.1.1 The first attempt

The first attempt suggested to clean up grep code, focused on the search.c file, which searches subroutines using dfa, kwset, and regex for grep. As of that attempt in November 2004, grep was at version 2.5.1.

The first mention of refactoring was in the form of a patch attached to an email message; we could not find discussion of this attempt before its contributor provided it. The patch submitter noted that the only problem with the patch was,

> “it relies on mempcpy, which is not always available, so this will have to wait until gnulib support is brought up to date”

The patch was discussed through a total of 16 messages among 8 grep contributors. 8 grep contributors in that email thread showed agreement with cleaning up search.c file by factoring out functions, however, they didn’t agree with two issues regarding how to clean up the file: the introduction of a mempcpy library function, and the use of a string function with macro arguments that are constant strings versus the minimization of using string functions.

In that email thread, it was argued that mempcpy was not part of every C library, and thus it was a problem for portable code. Another issue with string functions was discussed through the email thread. The argument concerned the choice of programming language for grep, whether grep needs migration from C programming language to C++ programming language, as it was suggested to use std::string in C++ instead of strlen in C language. However, it was concluded with that such a major move from C to C++ should be done with full consensus. Since the patch arrived without pre-discussion and planning, that consensus had not been achieved.

That is to say, in this episode, despite code being available, grep contributors did not reach a consensus on how to clean up the code, although they agreed that grep would benefit from refactoring. This refactoring code was never committed.

4.1.2 The second attempt

The second attempt to restructure grep code was suggested by another grep contributor in March 2005, and at that time grep was still at version 2.5.1. Again, the refactoring attempt appears to have begun with the provision of a patch providing refactoring.

This second attempt focused on the grep.c file, which is the main source file for grep. The patch submitter noted that the patch aimed to make it more maintainable for the future and pointed out the problems with the grep codebase. For instance, the patch submitter argued that a lot of bugs were coming out of the design problems, and the project kept working on fixing bugs or adding new features that made design problem worse. Hence, heavy restructuring was suggested as a priority to make grep code more maintainable. The refactoring patch solely written by the submitter included restructuring recursion, removing code duplications, factoring out functions and merging them into one place, and moving functions and related data. According to the patch submitter, applying his or her refactoring patch would make grep.c shorter and clearer. In addition, recursion will be much more understandable, which makes it easier to implement new features.

The patch to restructure grep.c was discussed through a total of 9 messages among 4 grep contributors. In that email thread, it was first pointed out that the refactoring patch was a very big one, that it depended on pending patches, and it duplicated part of other pending patches. Other contributors in that ongoing argument noted instant benefits from providing bug-fixes:

> “my main goal for the short term is to fix as many correctness bugs as can be fixed without making very large (and therefore inherently dangerous) changes.”

In addition, a need to update to a newer version of a library (gnulib) was also discussed as part of work to be done. As to the updating library issue, the patch submitter noted that updating the library might not be that easy, and a maintainer also argued that the decision to update the library needed another dimension of effort to look at gnulib manual and some examples of use cases in several OSS projects. In brief, adapting grep infrastructure to GNU library appears to need additional resources such as more experiences or expertise with GNU library project such as gnulib. This highlights that attempts to reduce future effort by drawing on libraries meets the technical intentions of refactoring, but simultaneously implies organizational changes. While these include technical familiarity with the libraries, they also include a new organizational dependency on the people producing the library and an understanding of their working practices, including their rhythm of updating the library.

In addition, this attempt to restructure code prompted other contributor to request that the grep maintainer needed to devise a summary of roadmap towards future releases. In the draft of a plan for grep, the maintainer laid out tasks for future grep releases 2.5.X and 2.6.0, and expressed some concerns to invest his time to those tasks. Those tasks included, for instance, updating regex.c file from glibc, adding new features, and heavy refactoring were planned towards grep 2.6 version. Interestingly, grep contributors responded to that what tasks they could work on, and the submitter of the second refigactoring patch responded to that email thread about a plan for grep as follows:

```text
"=> 2.6.x
> ======
> The following should go here:
> [...]  
> - heavy refactoring.
```

I think XXX [other grep contributor] and me agreed on sharing this one [heavy refactoring], me coming from above
(grep.c and neighborhood) and him from below (search.c and neighborhood)

My part is under patch #3797”

However, before starting restructuring grep code that requires large changes in the codebase, some grep contributors wanted to fix old bugs and merge long-in-waiting patches because getting bugs fixed and released would motivate contributors to attempt to participate in the project. Through discussion about which patches can be merged quickly or hold off because of the need of restructuring the patches, some pending patches were merged and included in the release of grep 2.5.2.

In the meantime, the submitter of the second refactoring patch left the message that she or he would leave the project because of its slowness of making progress such as merging the patches. Thus, the explicit division of labor to restructure code was not actually carried out. As a consequence, his or her restructuring patches remain uncommitted. As to that notification of his or her leave, another grep contributor mentioned that there seemed to be a general consensus on that a lot of grep code needs a complete rewrite, however, frequent bug-fix releases were essential for immediate users’ benefits, and code stability must be considered. Otherwise, grep might need another branch to experiment.

As a result, the second attempt is found to remain incomplete, when we tracked the grep patch tracking system. The original patch for refactoring were split into three patches, which reflects other’s suggestion to split that one big patch into smaller ones. However, those patches were labelled as “Need information”, “Won’t do”, and “Duplicate” by another grep maintainer, when heavy refactoring was actually carried out towards grep 2.6.

In brief, within the second episode we find that grep contributors again agreed with that grep needs restructuring work, and further, there was an explicit division of labor with a consensus on what should be done towards future releases. However, it appears that grep contributors deferred refactoring work because refactoring does not provide immediate benefits to grep users unlike frequent, quick bug-fixes or new features. Rather, refactoring implied risks, both in introducing new bugs and because the plan of work required completion of all tasks before seeing the anticipated benefits.

Taken together, in both episodes we note that each earlier attempt to refactoring was initiated and the refactoring patch was solely written by an individual contributor. Yet the patches did not complete the refactoring, but required substantial future effort from others. In short, the refactorings were presented as “gifts” but also functioned as “claims” on the future work of others. In both episodes, such attempts prompted other contributors to engage in argument about how to restructure code with which refactoring approaches. While grep contributors did not reach a consensus on how to restructure code in detail, we note that proceeding refactoring work needed resources such as time, and more experiences with adaptive maintenance such as adapting grep infrastructure with GNU library. In addition, the discussion indicated that there was concern about the status of “pending patches.” These concerns reflect uncertainty about the status of the codebase and thus the appropriate starting point for refactoring.

In summary, we find two attempts remained incomplete because the refactoring implied changes in organizational work practices that were beyond the existing commitments of the grep developers in the community-based OSS project.

### 4.2 Large-scale refactoring towards grep 2.6

While earlier two attempts to refactoring remained incomplete, we note that large-scale refactoring work between grep 2.5.4 and grep 2.6 was mostly carried out by two new grep maintainers over four months, from November 2009 to March 2010.

While there was no public email message that notified how two new maintainers were assigned or how they agreed with how to work towards grep 2.6 as of this writing, the email message in November 2009 announced that two new maintainers were assigned and noted that they had been involved in various GNU projects for many years and brought a lot of expertise to the GNU grep project. In addition, it also publicized what they had started working on: removing intl directory, cleaning up m4 directory, renaming, and adapting infrastructure.

We identified that the following tasks were carried out, based on the analysis of commit logs made on git between grep 2.5.4 and grep 2.6: reducing code duplications, eliminating unnecessary codes, moving source files, updating library files, removing old macro directories, fixing bugs submitted by previous contributors, and merging the pending patches (thus updating the overall status of the project before beginning).

In addition, we also note a large number of commits were made within a relatively short amount of time (i.e., over four months), based on the analysis of grep’s git. The analysis of git reveals that a total of 1593 commits were made to grep and a total of 39 developers made commits since November 1998. While the average commits per release is 72.36, we note that a total of 216 commits were made between grep 2.5.4 and grep 2.6 and especially, 207 commits were made by two new maintainers. 111 commits and 96 commits from each maintainer, respectively, towards grep 2.6 release, while the rest of 9 commits were contributed by other 6 grep contributors. This reflects a significant increase in the rate of work; not only did the refactoring developers work more, they worked more quickly than the usual pace of the project.

To get insights about architectural changes in GNU grep, we obtained the directory structure, and a dependency graph, using Scitool (see Figure 1). Figure 1 shows the structure of the architecture of grep 2.5.4 and grep 2.6. As can be seen in Figure 2, after large-scale refactoring, GNU grep 2.6 has only three directories in it.

The analysis of commit logs and the interview with one of grep 2.6 maintainer revealed that GNU grep switched to importing gnulib towards grep 2.6. In addition, we found that, as of grep 2.5.4 and grep 2.6, importing gnulib should’ve been manually done, and two new maintainers brought man power to carry it out because they already had worked on gnulib projects before joining GNU grep.
In addition, the messages between those two new maintainers publicly notified what tasks had been done by each of them, the patches to be pushed, how he or she had fixed old bugs and why some patches or bug-fixes would not be merged to git. That is to say, two new maintainers publicly, directly exchanged messages, while working on refactoring and other related tasks, which was laid out in a plan for grep March 2005.

In addition, one maintainer posted a message to ask whether there were any other patches or bugs, as grep 2.6 was planned to be released shortly. In that email thread, another maintainer listed what had been done towards grep 2.6 and who did what:

“4) check in the patches for the sync of dfc.c with GNU awk
   XXX [another grep maintainer] is doing it.
   ….
   7) some minimal cleanup of the grep(), grepdir(), recursion
   and fix--directories--read
   Planned by XXX [another grep maintainer]
   *-i --o
   *--color -i
   Patch posted by me [grep maintainer]”

That is to say, we find that two new grep maintainers shifted from the usual flow of work in grep to coordination mechanisms more typically associated with hierarchical organizations, such as explicit division of labor and direct communication in proceeding refactoring and other related tasks towards grep 2.6. The finding reveals that there was a concentrated effort on refactoring work between two new maintainers over four months and it appears they performed systematically by making decisions on who would work on adaptive maintenance and who would work on perfective maintenance.

4.3 Across episodes: What was different?

Our finding revealed that earlier two attempts to restructure code were unsuccessful due to uncertainty about the starting point for refactoring, time constraints. Further, the attempts implied organizational changes that others were unwilling to make. Activity continued but focused on tasks that could be accommodated by existing organizational structures, such as incremental bug-fixes.

In contrast, the third episode revealed significant organizational change proceeding attempts at refactoring. There was a concentrated, coordinated effort by two new maintainers for heavy refactoring and other related tasks. Considering those characteristics of earlier refactoring attempts, we sought to identify how heavy refactoring could be done by two new maintainers within a relatively short amount of time and what was different in the third attempt.

Taken as a whole, our finding suggests that organizational change in grep development was a factor that helped make it possible for two maintainers to work on large-scale refactoring, shifting to traditional coordination mechanisms: explicit division of labor, firm co-membership outside the OSS project, and direct, interpersonal communication outside the project.

To see whether there was external or corporate participation in refactoring work towards grep 2.6, we looked for commit logs made by those new maintainers and it was found that one of two maintainers used an affiliation email address, when making commits. The use of corporate organization’s email address indicates that his or her grep work is seen to (in)directly get paid or supported by the corporation. Another maintainer, who also worked on grep refactoring task, did not use affiliation email address, however, we notice that he was also working for the same corporation at that time, when we collected and analyzed the data of grep developer profile from the secondary source in business-oriented social networking services.

The interview discovered that those two maintainers worked for the same company during those periods of time, and exchanged private emails a lot, while working towards grep 2.6. Such individuals who share organizational ties have easy access to each other, and thus, firm co-membership outside the OSS project suggests that it was much easier to share information about solutions related to different parts of grep code. In contrast, the interview with a grep developer who maintained grep 2.5.4 reveals that all communication was made via public mailing list and his or her work was voluntary.

The evidence gleaned from multiple sources suggests large-scale refactoring that touches different parts of software application requires organizational change and implies organizational capabilities that are more common in hierarchical organizations used to plan interpersonal dependencies. The firm co-membership outside the OSS project helps explain about how the developers could overcome substantial costs of effort, expertise, and time, which is difficult to achieve by depending on independent, voluntary motivation, porous membership, and informal relationships.
5. An exploratory quantitative analysis: The impact of Software Refactoring

To examine the impact of refactoring on both technical and organizational dimensions, software metrics and developer base were analyzed. In this phase, we conducted an exploratory quantitative analysis.

5.1 Software metrics in two successive versions of grep

Table 1 compares the characteristics of grep prior to and immediately after refactoring. We plot the evolution of grep’s size and average cyclomatic complexity against release sequence numbers, following Lehman’s suggestion [23]. (Figure 2(a) and 2(b)). A total of 341 commits towards grep 2.5.1 in Figure 2(c) include the number of commits made from the initial version of grep because all earlier grep versions were not publicly committed to git.

The results show the striking effect of refactoring grep code. Lines of Code (LoC) decreased 71.78% from 31,313 LoC to 8,831 LoC, and the number of source files also decreased 74.41% from 86 source files to 22 source files. Software structural complexity, which was measured using each grep version’s average McCabe complexity, has reduced afterwards refactoring from 12.55 to 9.28. This indicates that grep complexity has changed from moderately complex to simple, based on the category of cyclomatic complexity range [6].

5.2 Developer base after refactoring

Figure 2(d) visualizes the evolution of the total number of grep developers who made commits (i.e., grep authors) and Figure 2(c) plots the total number of commits over time.

It is insightful to look at the increasing trend in the total number of grep authors after refactoring. The number of grep authors was between 1 and 5 from the initial version and grep 2.5.4 version. However, it increased up to 9 in grep 2.7.0 version and stays between 7 and 9 from grep 2.6.0 to 2.11.0 version. Although the number falls to between 2 and 4 from grep 2.12.0 version, a sudden increase in the number of authors appears in grep 2.15.0 and grep 2.17.0.

The change in grep developer base after refactoring suggests refactoring code appears to foster more grep developers. It
appears refactoring work that reduced the complexity of grep made grep code relatively much easier to comprehend, modify, and add new features. Accordingly, developer base seems to have increased after refactoring over time.

<table>
<thead>
<tr>
<th>Table 1. Metrics of grep 2.5.4. (before refactoring) and grep 2.6. (right after refactoring)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Metric</td>
</tr>
<tr>
<td>Lines of Code (LoC)</td>
</tr>
<tr>
<td>Lines of code in Library folder</td>
</tr>
<tr>
<td>Number of source files</td>
</tr>
<tr>
<td>Number of functions</td>
</tr>
<tr>
<td>Inactive Lines</td>
</tr>
<tr>
<td>Preprocessor Lines</td>
</tr>
<tr>
<td>Average Cyclomatic complexity</td>
</tr>
<tr>
<td>Cyclomatic complexity category</td>
</tr>
<tr>
<td>Max Cyclomatic complexity²</td>
</tr>
<tr>
<td>Max Nesting³</td>
</tr>
</tbody>
</table>

6. Discussion
The distinctive contribution of our study is that we richly illustrated how OSS production community could carry out large-scale refactoring that required direct, intense coordination and large effort, while organizational circumstances of OSS production is often characterized as voluntary, porous membership, and lack of formal authority. We also illuminated the factors that constrained or enabled software refactoring activities by identifying the differences across three episodes in the GNU grep project—two unsuccessful episodes and one successful episode. Our findings illustrated that two new grep maintainers who performed large-scale refactoring shifted towards traditional coordination mechanisms such as direct, interpersonal communication outside the project, explicit division of labor, and firm co-membership outside the project. They also ensured a stable, consensus starting point by updating the status of the project and avoiding devaluing the work of others by including the “pending patches” in their large-scale refactoring effort.

In other words, our study suggests large-scale refactoring requires changes in coordination mechanisms and organizational practice. Because making such large architectural changes requires to work across part of system in a large codebase, it creates new interdependencies among developers involved in refactoring in the large codebase. While organizational circumstances of OSS production communities are often characterized loosely-coupled, such changes in organizational forms of OSS production appears remarkable.

Then, our finding leaves us with an intriguing question: How could OSS production communities proceed large-scale refactoring in a way that does not interrupt open collaboration?

First, how could OSS contributors become committed to refactoring work over refactoring periods? The software refactoring literature has tended to assume that organizational circumstances are fixed at least in the short term. Particularly, it has assumed that dedicated developers or teams are always available during refactoring processes because they are bound to work to get paid. Accordingly, prior studies on software refactoring rarely considered how intensive coordination among sporadically available developers or teams could be orchestrated, while refactoring requires coordinated edits across part of systems and larger effort during refactoring period. Software developers in commercial organizations are bound to work on the assigned tasks, however, OSS contributors are independently motivated and free to come in and out of the project. In addition, in OSS production communities there is no manager who tells what should be done, assigns tasks to particular developers, or resource to pay developers who work on refactoring. The second attempt to restructure grep code demonstrate such a case that couldn’t proceed, as the contributor who showed the will to work on refactoring the code left the project.

Second, there is a risk of losing current OSS contributors, if fixing bugs and merging new or pending patches are delayed during refactoring periods because large-scale refactoring needs to freeze the code during refactoring periods. A new branch for refactoring in parallel can provide a solution, however, the Bibdesk 2.0 episode illustrates that OSS contributors returned to the un-refactored codebase to make incremental contributions because those were blocked by the refactoring freeze [17]. In the meantime, the un-refactored codebase becomes harder to be merged to the codebase in the process of refactoring. While Bibdesk may maintain their user base, refactoring couldn’t go further. In case of grep contributors in the second episode of our study, the project proceeded by merging pending patches and bug-fixes rather than refactoring because it provides immediate benefits to grep users in the short term, leaving large refactorings as goal deferred to a possible, but uncertain, future. Accordingly, refactoring open source code leaves us with potential concerns regarding how to maintain user bases during refactoring periods. Otherwise, new feature suggestions, bugs to be fixed, and pending patches likely stack up during refactoring. Even after refactoring, those tasks are unlikely easy to be merged to the refactored codebase, given the previous survey report that it’s hard to merge code after refactoring [21]. In short, it appears that refactoring is likely to succeed only if it occurs in a short, high activity, densely interconnected fashion; yet since projects don’t usually proceed on that basis, organizational change is necessary.

7. Limitations and Future work
Although our findings about OSS refactoring practice and the benefits of refactoring in both technical and organizational dimensions make important contribution, this study is not without limitations.

First, this study conducted a single case study with an exploratory approach, and thus, it inherently suffers from the representativeness of OSS projects. Given the finding that the vast majority of OSS projects are either small or medium size, and only a minor fraction are large [40], the size of the GNU grep project is seen appropriate to investigate community-based
FLOSS projects. However, GNU grep is a command-line utility, while a various types of utilities exist. To make our finding more generalizable, more community-based projects that went through large-scale refactoring needs to be collected and analyzed.

Second, we mainly focused on large-scale refactoring that touches different parts of software system. Hence, the focus of our study is not on regular small-scale refactoring, which happens in daily software practice. Given the granularity of refactorings would vary among researchers or practitioners, our finding will not be generalizable to other cases which went through small scale refactorings such as regular code reformatting or code cleanup that were observed in some OSS projects [16] and that appear in Fowler’s catalog [11].

Third, we focused on the change of software complexity to gauge software quality, rather than other metrics such as the number of bugs reported and time to be taken to fix the bugs right after software quality, rather than other metrics such as the number of bugs reported and time to be taken to fix the bugs right after refactoring. Thus, future work also calls for multidimensional assessment of refactoring with various complexity measures. For instance, in-depth quantitative study of bugs right after refactoring needs to be performed to assess the magnitude of refactoring benefits in the codebase.

8. Conclusion

Open collaboration has been of great interest because of its success despite well-known challenges such as fluid, porous membership, geographical dispersion, and no formal authority [24].

Our study hopes to make a contribution to the literature on open collaborations and software refactoring research community by presenting how organizational circumstances were shifted to perform tightly-coupled work such as large-scale refactoring, and the benefits of refactoring open source code on codebase as well as developer base over time. In our investigation of GNU grep, software refactoring remarkably reduced the structural complexity of OSS, and an increasing trend in developer base after refactoring. Prior studies found that decreased complexity is found to increase developer base [29], and purposeful redesign effort on Mozilla’s architecture aimed to make Mozilla more attractive contributors and it resulted it much more modular architecture of Mozilla [26]. While those previous research did not directly measure the impact of refactoring on the codebase [29] and developer base [26], our investigation of GNU grep highlights heavy refactoring in large code bases was successful in both technical and organizational dimensions.

As software projects make progress over time, they become inherently complicated, which makes it difficult to comprehend and modify [10]. The complexity of software becomes inherently grow over time. Considering the relationship between software complexity and maintainability [3], maintaining open source code and sustainable levels of OSS developers will become a larger concern in the long term. Given that refactoring is one of software engineering techniques to reduce software complexity, software refactoring can be one of solutions to control OSS complexity and to increase maintainability in the long term. OSS production communities could consider our findings about refactoring as one solution to improve open source quality and ability to maintain sustainable levels of developers who can write code, however, they need to gauge trade-offs between proceeding large-scale refactoring and users’ benefits from providing immediate bug fixes and merging new patches.

Our study opens the door to a lot of intriguing future work about how OSS work that requires substantial changes in the codebase is actually done in order to make open source code more maintainable, less complicated, and to improve OSS performance, given that it requires organizational changes by drawing on substantial costs of effort, time, and man power. Another stream of refactoring research found that incomplete or incorrect refactors cause bugs [14]. It was also found that a high ratio of refactoring is often followed by an increasing ratio of bug reports [36]. Those findings that refactorings induce bugs leave us intriguing research questions about how to ensure the quality of refactoring open source code and ongoing substantial costs of coordination right after large-scale refactoring.

9. REFERENCES


