On the Nature of Commits

Lile P. Hattori and Michele Lanza
REVEAL @ Faculty of Informatics - University of Lugano, Switzerland

Abstract

Information contained in versioning system commits has been frequently used to support software evolution research. Concomitantly, some researchers have tried to relate commits to certain activities, e.g., large commits are more likely to be originated from code management activities, while small ones are related to development activities. However, these characterizations are vague, because there is no consistent definition of what is a small or a large commit. In this paper, we study the nature of commits in two dimensions. First, we define the size of commits in terms of number of files, and then we classify commits based on the content of their comments. To perform this study, we use the history log of nine large open source projects.

1 Introduction

The history log of versioning systems, such as CVS or Subversion, stores a log for every change committed into the repository. Thus, it represents a rich amount of information about the changes of a software system. Because of their open and accessible nature, researchers often analyze versioning system history logs instead of processing all revisions of each file in a software system. The MSR (Mining Software Repositories) community is widely using this approach to investigate evolutionary aspects of software systems. Work in this area has dealt with correlating refactorings with defects by mining the comments of commits (Ratzinger *et al* [14]), investigating developer expertise based on their commit activities (Schuler and Zimmermann [16]), or detecting logical coupling by analyzing the frequency in which files changed together (Gall *et al.* [4]).

Although some studies characterized groups of commits, researchers misleadingly make assumptions like "large commits are outliers and, therefore, must be ignored". These assumptions are based on the observation that small commits occur more often, while large commits are rare exceptions. However, to the best of our knowledge, there is no study that tried to classify commits with respect to their size, with the exception of two approaches dealing with special cases: Purushothaman and Perry [13] used Swanson's classification of maintenance activities to analyze very small changes, and Hindle *et al.* [8] performed a similar study for large commits.

Our goal is to study commits to answer some seemingly simple questions: What is a small/large commit? Do development activities appear mainly in small commits? Are large commits only related to code management? To answer these questions we analyzed a total of 72,351 commits from nine open source projects. The results show that the distributions of commits from all 9 projects follow a Pareto distribution. These findings confirm previous observations [13] that most of the commits concern few files, whereas large commits are scarce. However, contrary to what one may assume, neither small commits are related exclusively to development activities, nor large commits are exclusively derived from management activities. Our study indicates that corrective actions, such as bug fixes, generate more tiny commits than any other activity. In addition, the number of large commits related to development is remarkably high, which demystifies the assumption that large commits are mainly related to management. The contributions of this paper are:

- A study of the statistical distribution of the size, in terms of changed number of files, of commits.
- A lightweight approach to classify each commit into development (forward engineering) or maintenance (reengineering, corrective engineering, management) activities, based on the content of their comments.
- A characterization of the commits of 9 open-source projects by evaluating their size and content distributions.

Structure of the paper. Section 2 presents related work that studied commits of software systems. Section 3 describes a study of the statistical distribution of size of commits and presents a size classification. In Section 4 we propose a content classification based on keywords that appear in a commit's comment. In Section 5 we present a case study that assesses both classifications using 9 open source projects according to the size and comment. Finally, Section 6 presents some conclusions and future work.

2 Related Work

Many researchers have explored the history logs to understand changes to a software system during its life cycle. Here we detail the work mostly related to ours.

Mockus and Votta [11] used word frequency and semantic analysis techniques to classify comments of commits according to Swanson's [17] classification of maintenance activities. Their classification consists of the following keywords: *corrective* – problem, incorrect, correct; *adaptive* – new, modify, update; and *perfective* – cleanup, unneeded, remove, rework. They compared the distribution of each maintenance activity with the size of a change in terms of lines added and deleted, and found that adaptive and inspection changes added most lines while inspection activities deleted many more lines than other activities.

Purushothaman and Perry [13] used the same classification to analyze very small changes. Their definition of small changes was: one or more modifications to single/multiple lines, one or more new statements inserted between existing lines, one or more lines deleted, or a modification to a single/multiple lines accompanied by insertion and/or deletion of one or more lines. One important finding of their study was that there is less than a 4 percent probability that a one-line change will introduce a fault in the code.

Later, Livshits and Zimmermann [10] used this observation as one of some data mining rules to find common error patterns by combining revision history with source code information. Hindle *et al.* [8] performed a similar study to Purushothaman and Perry on large commits. Selecting the 99th percentile of commits with respect to the number of files from nine projects, they manually classified them according to an extended version of Swanson's classification. They contrasted their findings against previous work by Purushothaman and Perry, and concluded that large commits are more perfective while small commits are more corrective.

Another work that classifies history logs is the case study conducted by Ayari *et al.* [2]. They collected Mozilla's CVS and Bugzilla information to evaluate the difficulties faced to integrate them. The case study pointed out that Mozilla's bug tracking database contains 50% of entries that are not related to corrective maintenance.

Research that correlates size of commits, and consequently, size of changes with project's activities has concentrated only on specific cases. Furthermore, they tend to present and use only relative values, such as 1% of commits that contain larger number of files. How large are these commits? Do they contain 30 or 1,000 files? What is the range? What is the percentile of commits with one, 10, 200 files? There is a need for an overall view of the distribution of development and maintenance activities according to the size of a change.

3 Size Classification

The size of a commit can be measured by the number of files involved or the number of added/deleted lines. In a previous study Herraiz *et al.* [7] showed that, at least for open source software, the evolution patterns in both counting source lines of code (SLOCs) or files is the same. We measure the size of a commit by counting the number of files it affects.

What is a "small" commit? There is no widely accepted classification of commits with respect to their size. For example, previous work classified small commits as those with one line changes, or large commits as the 99th percentile of all commits. In this work we segment commits into four groups based on a study of the statistical distribution of commits from nine different open source projects.

Project	Languages	Interval	Files	Commits
aMSN	C, C++, Tcl	28.05.02 - 04.06.08	10,080	10,092
ArgoUML	Java	26.01.98 - 29.10.07	11,631	12,334
Firebird	C, C#, C++, Java, Object Pascal, Python	26.06.06 - 23.05.08	3,941	580
JEdit	Java	16.01.00 - 27.05.08	15,837	12,714
JHotdraw	Java	12.10.00 - 10.05.07	5,237	391
Mantis	PHP	28.11.00 - 22.05.08	3,070	5,297
Miranda	C, C++, Delphi/Kylix, PHP	28.05.02 - 04.06.08	7,868	7,848
Spring	Java	17.06.05 - 28.05.08	41,258	12,596
Swig	C, C++	03.08.99 - 26.05.08	10,766	10,499

Table 1. Open Source Projects Analyzed

Table 1 shows the chosen projects, the programming languages used, the initial and final dates used in our analysis, the number of files in the last version considered and the total number of commits analyzed. To guarantee a good generalization of open source projects, we have selected projects with ages ranging from two to almost nine years, and that are written in various programming languages.

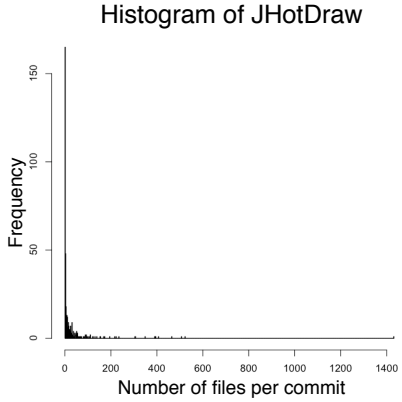

Figure 1. Histogram of JHotDraw

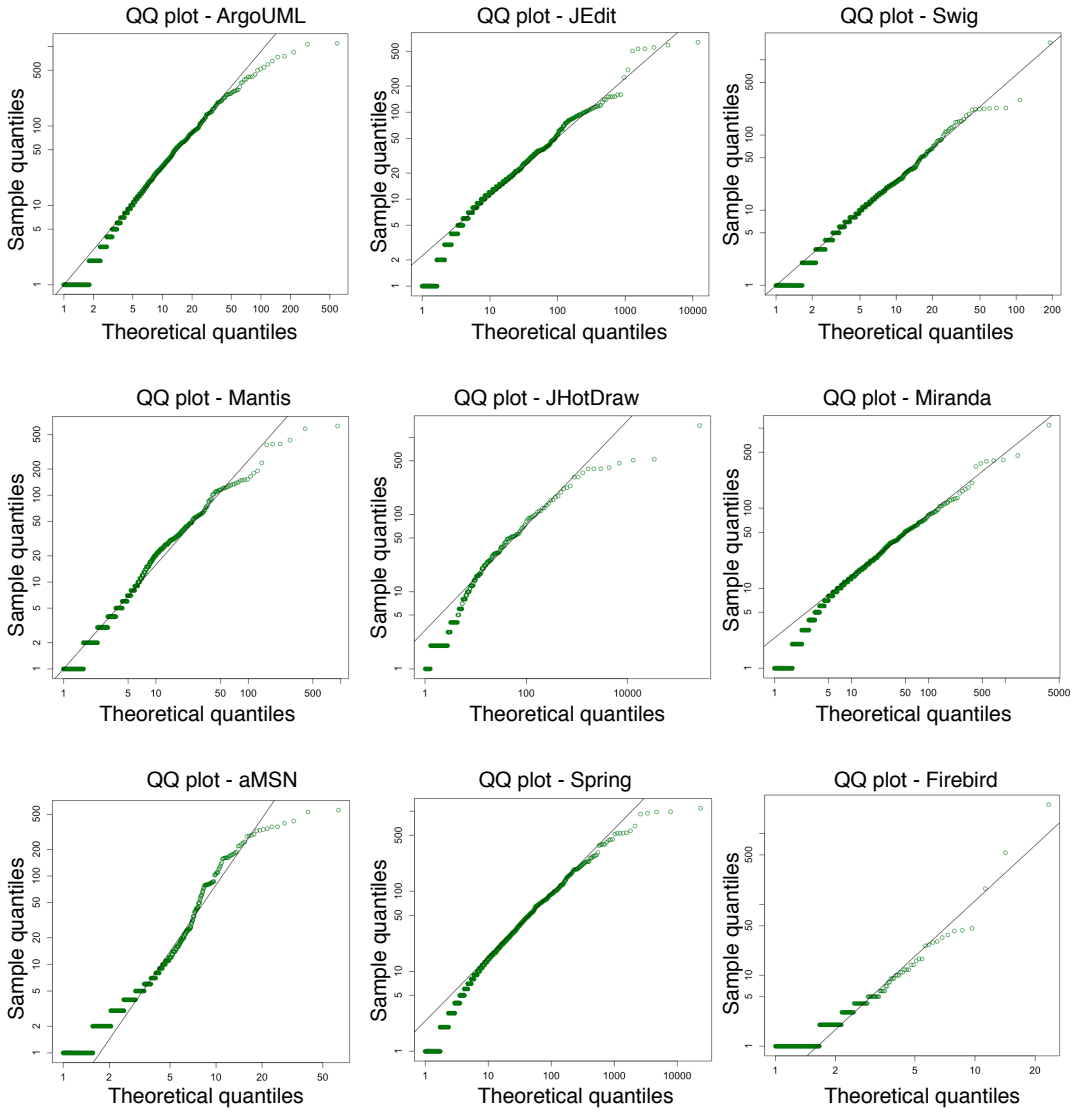

Figure 2. Quantile-Quantile Plot of each project's sample against a Pareto distribution

The first step to segment the commits is to investigate their distribution. Figure 1 shows the histogram of commits of JHotDraw. The x-axis represents the number of files per commit; the y-axis shows the frequency of commits that has a certain number of files. There are a great number of commits with few files and very few commits with hundreds of files, which suggests a power law distribution. The same pattern is observed in the other eight selected projects. To confirm this, we fit the commits from each project to a Pareto distribution [3, 12] through a quantile-quantile plot. A quantile represents an equally divided part of the frequency distribution, which in this case is a frequency of commits with a certain number of files.

Figure 2 shows the quantile-quantile plots of the commits versus the theoretical quantile of a Pareto distribution. They are ordered from the oldest to the youngest project. The interpretation of a q-q plot is: if the population distribution is the same as the comparison distribution this approximates a straight line, especially near the center. As it can be observed, almost all q-q plots approximate a straight line, which confirms that they follow a Pareto distribution. Only the q-q plot of Firebird shows a slightly different pattern, because it has too few large commits (to be more precise, only three commits have more than 125 files). This behavior can be due to the age and the number of commits. It is the youngest project, aging less than two years, and it has the second fewest number of commits registered. So, one hypothesis to explain this result is that this project is still under active development, a phase that usually generates more small commits.

Our goal is to classify commits into a restricted number of significant groups. Since commits follow a Pareto distribution, it does not make sense to split them into quartiles, for example, because the number of commits with only one file is around the 50th percentile in most cases. Although we could use the approximate distribution function found for each project to calculate an exact division, this is not a generalized approach that could be directly applied to other open source projects.

The approach we propose is to divide the commits into fours groups by using an exponential scale. Although the exponential scaling parameter for power law distributions typically lies in the range $2 < \alpha < 3$ [3], we choose 5 as exponential scaling parameter. Otherwise the last group would range from 16 or 81 up, which would still be a small number compared to some commits with hundreds of files in it. The proposed size classification of commits is:

- **tiny:** 1 to 5;
- **small:** 6 to 25;
- **medium:** 26 to 125;
- **large:** 126 up.

This classification is used in section 5 together with the content classification, presented next, to reason about the nature of commits.

4 Comment Classification

The comment of a commit is a textual message usually related to the activity that generates the new piece of code that is being committed. It ranges from a simple note to a detailed description, depending on the project's conventions and on the developer's behavior. Although there is no universal convention on writing a comment, some words frequently appear to describe an activity, *e.g.,* the words "fix" and "bug" when someone is fixing a bug.

Comment keywords were previously used to classify small and large changes according to Swanson's classification of maintenance activities [8, 13]. We could adopt the same approach if we were not analyzing open source software. Previous works have proven that the growth of open source software is different from commercial software systems. While commercial systems tend to have a sublinear growth when reaching the maintenance phase, open source software tend to have linear to superlinear growth [5, 6, 7]. This implies that open source software is continuously evolving, and incorporating new features, rather than only being maintained.

For this reason, we propose a different classification with four major activities: *forward engineering* as a development activity; and *reengineering*, *corrective engineering* and *management* as maintenance activities.

Forward engineering activities are those related to incorporation of new features and implementation of new requirements. *Reengineering* activities are related to refactoring, redesign and other actions to enhance the quality of the code without properly adding new features. *Corrective engineering* handles defects, errors and bug in the software. *Management* activities are those unrelated to codification, such as formatting code, cleaning up, and updating documentation.

Development	**Maintenance**		
Forward Engineering	**Reengineering**	**Corrective Engineering**	**Management**
implement	optimiz	bug	clean
add	ajdust	fix	license
request	update	issue	merge
new	delet	error	release
test	remov	correct	structure
start	chang	proper	integrat
includ	refactor	deprecat	copyright
initial	replac	broke	documentation
introduc	modif		manual
creat	(is, are) now		javadoc
increas	enhance		comment
	improv		migrat
	design change		repository
	renam		code review
	eliminat		polish
	duplicat		upgrade
	restrutur		style
	simplif		formatting
	obsolete		organiz
	rearrang		TODO
	miss		
	enhanc		
	improv		

Table 2. Keywords used to classify comments

Table 2 presents the keywords selected for each activity. Some of them are only the radicals or parts of words. We collected these keywords by first manually analyzing the commits from ArgoUML and JHotdraw projects, and then interactively applying, refining and reapplying the keywords on the other projects. There is theoretically a fifth category of commit comments, the empty ones, which of course cannot be classified using our approach. To our surprise their frequency is relatively high, even in the case of larger commits. We discuss this point in the next section.

The algorithm we are currently using to classify commits simply searches for keywords and classifies a commit as soon as it finds any keyword in the text. It searches for comments in the following order: empty messages, management, reengineering, corrective engineering, and forward engineering. A commit can embrace changes from more than one activity, *e.g.,* a developer can do a clean up on a class while fixing a bug. For this case, our algorithm classifies the commit according to the first keyword found. We performed a case study to assess this approach, which is presented in the next section.

5 Experimental Study

In this experimental study we first assess the comment classification presented in Section 4 by manually classifying the commits of a period of three months of a Java and a C++ project, and compare it with the results from the automatic classification in terms of precision and recall. We then apply both size and comment classifications in the nine open source projects and reason about the results obtained.

5.1 Case Study

To evaluate our proposed comment classification, we manually classified a total of 1,088 commits from ArgoUML and aMSN, and compared them with the automatic classification in terms of precision and recall. Given a set of commits automatically classified A, and a set of manually classifies commits M, we define theses two measures as:

Precision is the proportion of commits automatically classified that was correct given the manual classification.

$$P = \frac{\mid A \bigcap M \mid}{\mid A \mid}$$

Recall is the proportion of commits manually classified that was correctly classified by our algorithm.

$$R = \frac{\mid A \bigcap M \mid}{\mid M \mid}$$

The intersection $A \bigcap M$ contains only commits that were equally classified by the two methods. This means that commits that had no classification assigned to it or commits wrongly classified by the algorithm were left outside the intersection. Finally, we calculate the F-measure from precision and recall to get an average result:

$$F = \frac{2 * P * R}{P + R}$$

For ArgoUML, we manually classified 601 commits and the algorithm was able to automatically classify 529 of them, with 395 correspondences. This gives a precision of 0.75 and a recall of 0.66, which means that 75% of the automatically classified commits were correct and 66% of the correctly classified commits were automatically classified. The F-measure was 0.70, *e.g.,* every 7 out of 10 commits were correctly classified. The result, in the light of the algorithm's simplicity is impressive and supports the fact that lightweight approaches are viable in this context. Since ArgoUML was initially used for choosing keywords, we needed to investigate if this result is an exception. Therefore, we applied the same measures on another project.

We chose aMSN, a project developed in a different programming language to avoid conventions used for a particular language. We manually classified 487 commits and the algorithm classified 415, with 345 correspondences. Precision was 0.83 and recall 0.71, which gives an F-measure of 0.76.

The effectiveness of our algorithm was proven by the results found for aMSN, being even superior to the ones found for ArgoUML. Even though the algorithm is suitable for this study, other approaches could be further investigated to increase precision and recall values, such as semantic analysis techniques [1].

5.2 Experimental classification

The classification of commits is presented in two different charts, depicted in Figure 3 and Figure 4.

Figure 3 presents bar charts containing the distribution of commits per size for each activity. Note that the y-axis is presented in log scale. These charts give an overall view of the absolute distribution of commits over the activities. For example, the chart of Mantis shows that the number of commits related to corrective activities is greater than the others for tiny, small and medium commits.

Figure 4 shows the empirical cumulative distribution function of commits for each activity. The vertical lines are divisions for size segments. These charts show the relative distribution of commits for each activity. For example, the chart of JHotDraw shows that approximately 35% of commits related to management activities are tiny. We first analyze the results for each project, individually, and then we present an overall assessment.

ArgoUML. The activities distribution is balanced for all sizes. In other words, for every size the number of commits assigned to one activity is close to the number of commits assigned to the others. The only visible exception is the number of large commits related to management, which is notably higher than the number of commits of other three activities. The great majority – around 80% – of classified commits is tiny.

JEdit. Like in ArgoUML, the activities distribution is balanced for all sizes. There is only one exception: there is no large commit related to reengineering. Around 70% of all classified commits are tiny, 25% are small, and less than 5% are distributed between medium and large commits.

Swig. In this project, the distributions are very balanced, both proportionally (Figure 4) and absolutely (Figure 3). However, there are no large commits related to reengineering and forward engineering.

Mantis. The empirical cumulative distribution functions for this project show little proportional difference among the activities. For example, there are more small and tiny commits for corrective engineering than for management.

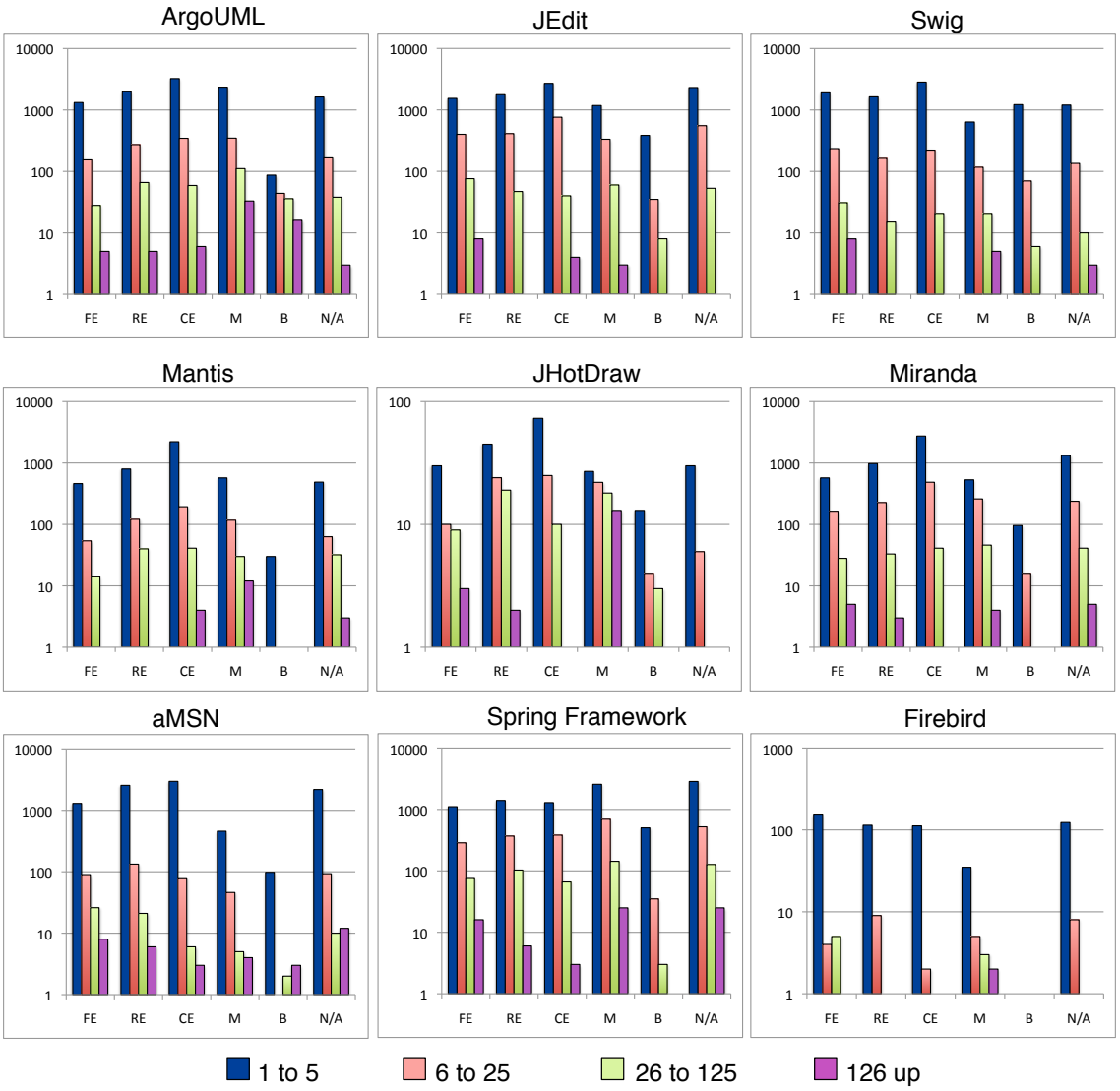

Figure 3. Classification of commits based on their size and comment's content. FE = Forward Engineering. RE = Reengineering. CE = Corrective Engineering. M = Management. B = Blank Comments. N/A = Unclassified Comments.

This is also true if we observe the absolute values, where we can see that commits for corrective engineering prevail for tiny, small and medium commits.

JHotDraw. Proportionally, there is a great difference among the activities for tiny commits, because 80% of commits related to corrective engineering are tiny, while there are only 30% of commits related to management. Forward engineering and reengineering are in between the previous two. This behavior is also found for medium and large commits. Looking at Figure 3, we understand why the proportional value is so low for tiny commits, compared to other projects. In this project, the number of large commits related to management is higher than large commits for other activities, which provokes the unusual shape of the corresponding empirical cumulative function.

Miranda. From the cumulative distribution function chart we notice that the proportion of commits related to forward engineering and reengineering is very similar. The number of commits related to corrective engineering is again notably high for tiny, small, and medium commits, while there are no large commits.

aMSN. In this project, the number of tiny commits is notably high. Around 95% of commits related to corrective engineering, 90% of commits related to reengineering,

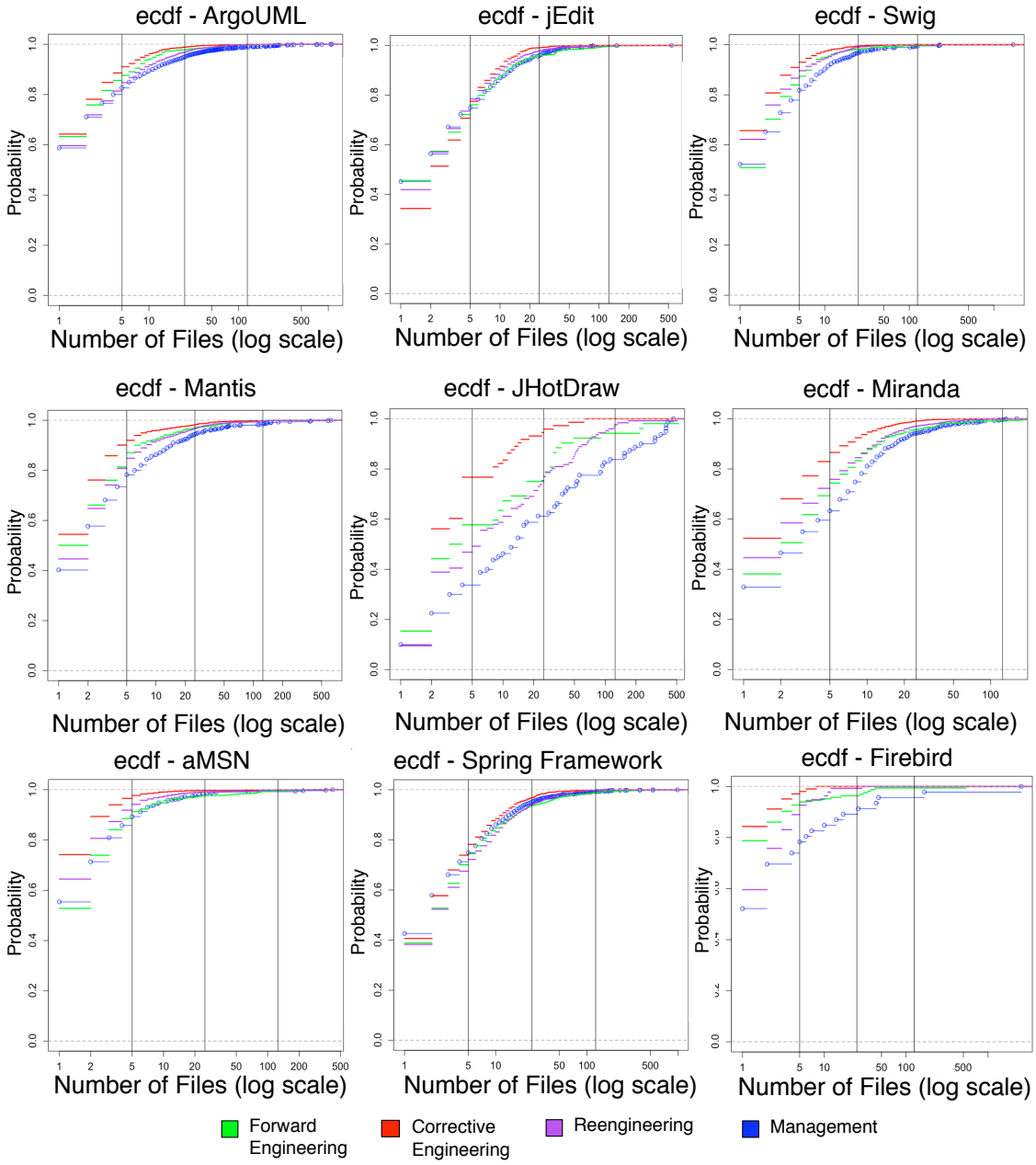

Figure 4. Empirical cumulative distribution functions of each activity for all projects

and 85% of commits related to management and forward engineering are tiny. This means that about 5% of commits related to corrective engineering are spread around small, medium and large commits. This is probably one case in which an exponent around 2 would be more suitable to split the size segments.

Spring. Similar to what happens with JEdit and ArgoUML, the proportion of activities is homogeneous for each size classification. The same conclusion is drawn when we analyze the bar chart.

Firebird. This is the project with the lowest number of commits. For this reason, medium and large commits only appear in forward engineering and management, while large commits appear exclusively in management. This can be easily observed in the two corresponding empirical cumulative distribution functions. For this reason, Firebird is also a project in which a low exponent seems to be more suitable to split the size segments.

Discussion. In general, tiny commits are approximately 80% of the total, small commits are 15%, medium commits are less than 5% and large commits are less than 1%. Blank comments appear in most projects, mainly associated with tiny and small commits, but sometimes also on larger commits. This finding is disturbing, as it hints at one of the reasons why evolution is hard to control. Only one project does not present blank comments: Firebird. Our algorithm was unable to classify less than 20% of all commits.

Tiny commits are more related to corrective activities, followed by forward engineering and reengineering, that alternate positions. Management activities occupy the last positions, with two exceptions: Spring, where they are the most numerous, and Mantis. The distribution of activities in small commits is heterogeneous. However, forward engineering was the least frequent activity in five projects. Medium commits are completely heterogeneous.

A surprising result appears in large commits: management activities are the majority in five projects, but forward engineering occupies the first position in four. In addition, both activities appear in all projects, while forward engineering and corrective engineering only appear in seven of them. With this result, we confirm the findings by Hindle *et al.* [8] that a great number of large commits is actually related to the development of new functionalities. This also points to one shortcoming of current versioning systems, where a developer has to explicitly commit: the time between two commits is completely arbitrary and could be very long, thus leading to a loss of fine-grained evolutionary information [9, 15].

Another interesting observation is that the proportion of management activities related to larger commits is greater than the proportion of the other three activities. This information appears in Figure 4: the management line is the lowest. Conversely, the proportion of larger commits is low for corrective engineering activities. Hence, we can state that management activities tend to generate larger commits, while corrective activities tend to be small and localized.

Threats to validity. Although we tried to diversify the characteristics of projects by carefully choosing nine different open source projects, some of the finding may not be generalized to other open source projects. In addition, these findings may not be generalized to industrial software system, since open source projects have particular characteristics, such as linear to superlinear growth rate [5, 6]. Another issue concerns the use of number of files as a measure of the size of a commit. Even though Herraiz *et al.* [7] have showed that in open source software SLOCs and number of files are equivalent for evolutionary patterns, the results found in our study may not be valid for the use of SLOCs to measure the size of a commit.

6 Conclusions and Future Work

In this work we studied the nature of commits by considering two aspects: the number of files involved in each commit, and the content of the log message.

We first investigated the distribution of commits according to the number of files and confirmed that it follows a Pareto distribution. This implies that the majority of commits have very few files in it, while few commits contain a large number of files. We also proposed size segmentation into four groups of commits: tiny, small, medium and large. For that, we used an exponential scale with value equals to five. This division suited most projects, except the aMSN project, from which more than 90% of the commits were classified as tiny.

We also classified the commits according to development and maintenance activities based on the content of their comments. Instead of using a previous classification of maintenance activities, we proposed a new classification that is more suitable for open source projects, according to a number of researches [5, 6, 7]. It takes into account that an open source project is constantly adding new features to the software and evolving, rather than only maintaining it.

Some important findings of this study are that the majority of tiny commits are not related to development activities. Corrective actions are the ones that generate more tiny commits. Development activities are spread among all sizes of commits. However, they appear with management as the two most common activities for large commits. Together with Hindle's work [8] we demystify the assumption that large commits are outliers, especially because they do not represent dots far away from one project's empirical distribution function.

Since the great majority of commits are very small and there is a significant number of large commits that actually change the source code, the probability that lots of small changes are committed in between the time a developer is performing a large change is high. In the case of Subversion and CVS, this implies that the developer who is performing a major change, *e.g.,* a structural change, will have to synchronize his changes with other small changes before committing then into the repository. Merging issues have been a common complaint among developers that could be avoided if versioning systems had a code-based mechanism for automatically synchronizing fine-grained changes.

For future work, we intend to correlate the roles of the developers with the size of the commits and the type of comment. Moreover, since the projects analyzed had initially used CVS as their versioning system and migrated recently (from 2004 up) to Subversion, we would like to investigate projects that used Subversion since the beginning. Subversion groups together files in a commit, while CVS commits each file individually. Although Subversion tries

to reconstruct these commits when they are parsed from CVS, there might be remarkable differences, like fewer tiny commits in a project that uses Subversion.

Acknowledgments. We gratefully acknowledge the financial support of the Swiss National Science foundation for the project "REBASE" (SNF Project No. 115990).

References

[1] Feature location using probabilistic ranking of methods based on execution scenarios and information retrieval. *IEEE Trans. Softw. Eng.*, 33(6):420–432, 2007. Denys Poshyvanyk and Yann-Gael Gueheneuc and Andrian Marcus and Giuliano Antoniol and Vaclav Rajlich.

[2] K. Ayari, P. Meshkinfam, G. Antoniol, and M. D. Penta. Threats on building models from cvs and bugzilla repositories: the mozilla case study. In *CASCON '07: Proceedings of the 2007 conference of the center for advanced studies on Collaborative research*, pages 215–228, New York, NY, USA, 2007. ACM.

[3] A. Clauset, C. R. Shalizi, and M. E. J. Newman. Power-law distributions in empirical data, Jun 2007.

[4] H. Gall, M. Jazayeri, and J. Krajewski. Cvs release history data for detecting logical couplings. In *IWPSE '03: Proceedings of the 6th International Workshop on Principles of Software Evolution*, page 13, Washington, DC, USA, 2003. IEEE Computer Society.

[5] M. W. Godfrey and Q. Tu. Evolution in open source software: A case study. In *ICSM '00: Proceedings of the International Conference on Software Maintenance (ICSM'00)*, page 131, Washington, DC, USA, 2000. IEEE Computer Society.

[6] I. Herraiz, J. M. Gonzalez-Barahona, and G. Robles. Towards a theoretical model for software growth. In *ICSEW '07: Proceedings of the 29th International Conference on Software Engineering Workshops*, page 21, Washington, DC, USA, 2007. IEEE Computer Society.

[7] I. Herraiz, G. Robles, and J. M. Gonzalez-Barahon. Comparison between SLOCs and number of files as size metrics for software evolution analysis. In *CSMR '06: Proceedings of the Conference on Software Maintenance and Reengineering*, pages 206–213, Washington, DC, USA, 2006. IEEE Computer Society.

[8] A. Hindle, D. M. German, and R. Holt. What do large commits tell us?: a taxonomical study of large commits. In *MSR '08: Proceedings of the 2008 international workshop on Mining software repositories*, pages 99–108, New York, NY, USA, 2008. ACM.

[9] M. Kersten and G. C. Murphy. Using task context to improve programmer productivity. In *SIGSOFT '06/FSE-14: Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 1–11, New York, NY, USA, 2006. ACM.

[10] B. Livshits and T. Zimmermann. Dynamine: finding common error patterns by mining software revision histories. *SIGSOFT Softw. Eng. Notes*, 30(5):296–305, 2005.

[11] A. Mockus and L. G. Votta. Identifying reasons for software changes using historic databases. In *ICSM '00: Proceedings of the International Conference on Software Maintenance (ICSM'00)*, page 120, Washington, DC, USA, 2000. IEEE Computer Society.

[12] M. E. J. Newman. Power laws, pareto distributions and zipf's law. *Contemporary Physics*, 46:323, 2005.

[13] R. Purushothaman. Toward understanding the rhetoric of small source code changes. *IEEE Trans. Softw. Eng.*, 31(6):511–526, 2005. Dewayne E. Perry.

[14] J. Ratzinger, T. Sigmund, and H. C. Gall. On the relation of refactorings and software defect prediction. In *MSR '08: Proceedings of the 2008 international workshop on Mining software repositories*, pages 35–38, New York, NY, USA, 2008. ACM.

[15] R. Robbes and M. Lanza. Versioning systems for evolution research. In *IWPSE '05: Proceedings of the Eighth International Workshop on Principles of Software Evolution*, pages 155–164, Washington, DC, USA, 2005. IEEE Computer Society.

[16] D. Schuler and T. Zimmermann. Mining usage expertise from version archives. In *MSR '08: Proceedings of the 2008 international workshop on Mining software repositories*, pages 121–124, New York, NY, USA, 2008. ACM.

[17] E. B. Swanson. The dimensions of maintenance. In *ICSE '76: Proceedings of the 2nd international conference on Software engineering*, pages 492–497, Los Alamitos, CA, USA, 1976. IEEE Computer Society Press.