What Do Large Commits Tell Us? A taxonomical study of large commits

Abram Hindle
David Cheriton School of Computer Science
University of Waterloo
Waterloo, Ontario
Canada
ahindle@cs.uwaterloo.ca

Daniel M. German
Department of Computer Science University of Victoria
Victoria, British Columbia
Canada
dmg@uvic.ca

Ric Holt
David Cheriton School of Computer Science
University of Waterloo
Waterloo, Ontario
Canada
holt@cs.uwaterloo.ca

ABSTRACT

Research in the mining of software repositories has frequently ignored commits that include a large number of files (we call these large commits). The main goal of this paper is to understand the rationale behind large commits, and if there is anything we can learn from them. To address this goal we performed a case study that included the manual classification of large commits of nine open source projects. The contributions include a taxonomy of large commits, which are grouped according to their intention. We contrast large commits against small commits and show that large commits are more perfective while small commits are more corrective. These large commits provide us with a window on the development practices of maintenance teams.

Categories and Subject Descriptors

D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement; D.2.9 [**Software Engineering**]: Management—*Life cycle, Software configuration management*

General Terms

Legal Aspects, Measurement, Verification

Keywords

Large Commits, Source Control System, Software Evolution

1. INTRODUCTION

What do large commits tell us? Often when studying source control repositories large commits are ignored as outliers, ignored in favour of looking at the more common smaller changes. This is perhaps because small changes are likely to be well defined tasks performed on the software system and perhaps it is easier to understand the intentions behind these changes, and draw conclusions from them.

MSR'08, May 10-11, 2008, Leipzig, Germany.

Yet, large commits happen. If we ignore them in our studies, what are we missing? What information do they contain that can help us understand the evolution of a software project? There is plenty of anecdotal information about large commits. A common cited cause for them is a massive copyright change, or reformatting its source code [4].

In this paper we analyze nine popular open source projects. We study two thousand of their largest commits in an attempt to answer two fundamental questions: what prompts a large commit, and what can we learn from such large commits.

Before this study, when asked what are large commits, we simply answered ancedotally, but now we can provide publicly available examples about large commits. We plan to provide a ranking of the frequency of the classes of commits we found by investigating many large commits from multiple OSS projects. This will allow future researchers and users to rely on both their own experience and the evidence provided in this paper rather than just their own ancedotal evidence. We also will describe the difference in purpose behind large commits versus small changes.

1.1 Previous Work

Swanson [10] proposed a classification of maintenance activities as corrective, adaptive and perfective, which we employ in this paper. Others [9] have characterized small changes and what they mean in the context Swanson's classification of maintenance, faults and lines of code (LOC).

The work exists within the context of the Mining Software Repositories community. Many of these studies extract information from source control systems such as CVS [3, 11].

Many studies have investigated specific projects in detail and have done deep in depth case studies of the evolution of specific OSS projects [2, 6]. Others have gone about quantifying and measuring change [8, 7, 5] in source control systems (SCS). Many software evolution researchers such as Capiluppi et al [1] and Mockus et al. [8] have studied multiple OSS projects from the perspective of software evolution.

2. METHODOLOGY

We have two primary research questions:

1. What are the different types of large commits that occur in the development of a software product?
2. What do large commits tell us about the evolution of a software product?

Software Project	Description
Boost	A comprehensive C++ library.
Egroupware	A PHP office productivity CMS project that integrates various external PHP applications into one coherent unit.
Enlightenment	A Window Manager and a desktop environment for X11.
Evolution	An email client similar to Microsoft Outlook.
Firebird	Relational DBMS gifted to the OSS community by Borland.
MySQL (v5.0)	A popular relational DBMS (MySQL uses a different version control repository for each of its versions).
PostgreSQL	Another popular relational DBMS
Samba	Provides Windows network file-system and printer support for Unix.
Spring Framework	A Java based enterprise framework much like enterprise Java beans.

Table 1: Software projects used in this study.

We used a case studies as the basis for our methodology. We selected nine open source projects (listed in table 1). These projects were selected based upon three main constraints: a) that they were at least 5 years old, mature projects, with a large user base; b) that their version control history was available; and c) that they represented a large spectrum of software projects: different application domains (command line applications, GUI-based, server), programming languages (PHP, C++, C, Java), development styles (company sponsored–MySQL, evolution–or community development–PostgreSQL).

The first stage (of two stages) of our research consisted in the creation of a classification of commits. We proceeded as follows:

1. For each project, we retrieved their commit history. We then selected the 1% commits, per each project, that contained the largest number of files (of any file type, not only source code) for our manual inspection. We auditted 2000 commits.

2. We created a classification of large commits. We used Swanson's Maintenance Classification [10] as the starting point. As its name implies, Swanson's Maintenance Classification is mostly oriented towards activities that adapt an existent system. We added two more categories to it: *Implementation* (adding features), and *Non-Functional*. Non-functional are changes to the software that are not expected to alter its functionality in anyway, but need to be performed as part of the typical software development cycle. For example, adding comments or reformatting source code. These are summarized in Table 2.

3. Based upon these categories of changes, and the issues that they address we proceeded to create a candidate list of types of commits we would expect to find. These can be seen as "low level" descriptors of the intention of the developer such as "add feature", "bug fix", "change of license", "reindentation". We refined this list by **manually** classifing the large commits of the first two projects we studied (MySQL and Boost). This classification helped us improve and refine our list of types, which is detailed in Table 3. We also discovered that a commit can be of one or more types (frequently a commit contains several independent changes). We then mapped these types of commits back into the Extended Swanson Classification, as shown in table 4.

Categories of Change	Issues addressed.
Corrective	Processing failure Performance failure Implementation failure
Adaptive	Change in data environment Change in processing environment
Perfective	Processing inefficiency Performance enhancement Maintainability
Implementation	New requirements
Non functional	Legal Source Control System management Code clean-up

Table 2: Extended Swanson Categories of Changes. They were used to identify types of commits. The first three are based upon Swanson Maintenance Classification.

Category of Change	Types of Change.
Corrective	bug, dbg
Adaptive	plat, bld, test,doc, data, intl
Perfective	cln, ind, mntn, mmod, rfact, rmod
Implementation	init, add, fea, ext, int
Non functional	lic, rmmod, ren, trpl, mrg
Other	cross, brch

Table 4: An attempt at classifying types of changes according to the Extended Swanson Categories of Change. Some of the types of change do not fit only one category; for example, a documentation change might be adaptive or perfective. See table 3 for a legend of types.

Type commit	Abbrev.	Description
Branch	brch	If the change is primarily to do with branching or working off the main development trunk of the version control system.
Bug fix	bug	One or more bug fixes.
Build	bui	If the focus of the change is on the build or configuration system files (such as Makefiles).
Clean up	cln	Cleaning up the source code or related files. This includes activities such as removing non-used functions.
Legal	lic	A change of license, copyright or authorship.
Cross	cross	A cross cutting concern is addressed (like logging).
Data	data	A change to data files required by the software (different than a change to documentation).
Debug	dbg	A commit that adds debugging code.
Documentation	doc	A change to the system's documentation.
External	ext	Code that was submitted to the project by developers who are not part of the core team of the project.
Feature Add	fea	An addition/implementation of a new feature.
Indentation	ind	Re-indenting or reformatting of the source code.
Initialization	init	A module being initialized or imported (usually one of the first commits to the project).
Internationalization	int	A change related to its support for languages other-than-English.
Source Control	scs	A change that is the result of the way the source controls system works, or the features provided to its users (for example, tagging a snapshot).
Maintenance	mntn	A commit that performs activities common during maintenance cycle (different from bug fixes, yet, not quite as radical as new features).
Merge	mrg	Code merged from a branch into the main trunk of the version control system; it might also be the result of a large number of different and non-necessarily related changes committed simultaneously to the version control system.
Module Add	add	If a module (directory) or files have been added to a project.
Module Move	mmod	When a module or files are moved or renamed.
Module Remove	rmod	Deletion of module or files.
Platform Specific	plat	A change needed for a specific platform (such as different hardware or operating system).
Refactoring	rfact	Refactoring of portions of the source code.
Rename	ren	One or more files are renamed, but remain in the same module (directory).
Testing	tst	A change related to the files required for testing or benchmarking.
Token Replace	trpl	An token (such as an identifier) is renamed across many files (e.g. change the name or a function).
Versioning	ver	A change in version labels of the software (such as replacing "2.0" with "2.1").

Table 3: Types of Commits used to annotated commits. These types attempt to capture the focus of the commit rather than every detail about it; for instance *Token Replacement* that affected 1 build file and 10 source files will not be labelled *Build*. A commit could be labelled with one or more types. Their abbreviation is used in the figures of this paper.

The Extended Swanson Classification appeared insuficient to classify large commits. We noticed that many of the large commits such as build (bld), module management (add, rmod, mmod), were difficult to place into these categories. In many cases the intention of the commit was none of these. For this reason we decided to create a new taxonomy, which we call *Categories of Large Commits*, which is summarized in table 5. We used this taxonomy to organize the types of commits, as depicted in Table 6.

Manually classifying commits is difficult. We are not contributors to any of these projects, and relied on our experience as software developers to do so. The procedure we used was the following:

- We read the commit log. Most of the commits in these projects have log comments which provide a good rationale behind the commit. In some cases the log was too explicit (for example, the largest log comment in Evolution was 13,500 characters long), but in many cases the commit was simple and clear in its intention (e.g. "HEAD sync. Surprisingly painless","Update the licensing information to require version 2 of the GPL").
- We identified the files changed. Usually the filename (and its extension) provides certain clues (e.g. *.jpg are most likely documentation or data files).
- We studied the `diff` of the commit, and compared its contents to the commit log. We believe that the contrasting of both sources of information improved the quality of our classification. If they appeared to contradict each other we used the information in the diff.

The result of this classification was, for each commit, a list of types of commits that reflected the intention of such commit.

For the second stage of our study we proceeded as follows:

1. For each project we retrieved (with the exception of MySQL[1]) the log for each commit, and the corresponding diff to its files.
2. We proceeded to **manually** classify each of these commits into one or more of the different types of large commits (as shown in Table 3). Every commit was labelled with one or more types.
3. For each project we created a summary of what we consider the "theme" of the large commits. In other words, we tried to draw a rational that explained why the project was doing large commits, and what such commits explained about the project, its organization, development process, or its developers (qualitative analysis).
4. We quantitatively analyzed the resulting data, both by project, and as an average over all projects.

3. RESULTS

We present the results of our study in two parts. In the first part we describe the themes of our qualitative results. In the second part we present an statistical overview of the types of large commits.

[1]MySQL uses `Bitkeeper` as version control system, and we were not able to retrieve the diffs for each commit.

Categories of Large Commits	Types of Commits
Implementation	fea, int, plat
Maintenance	bug, dbg, mntn, cross
Module Management	add, mmod, rmod, split
Legal	lic
Non-functional source changes	cln, trpl, rfct, indent
SCS Management	brnch, ext, merge, ver, scs
Meta-Program	bui, tst, doc, data, intl

Table 6: Classification of the types of commits using the Categories of Large Commits

3.1 Themes of the Large Commits

By reading and classifying the log-changes we were able to draw certain conclusions of the rationale behind many of the large commits. We refer to these conclusions as the *themes* of the large commits of each project.

Boost's documentation produced many large commits because they were auto-generated from the source code and docbook files (manuals). Boost developers used branches and consequently many of the large commits were merges from branches to the trunk. A common yearly large commits was an update to the year of the copyright. Boost also changed its license partway through development. Boost has a well-defined source code style and we observed many cleanup commits (reformatting and rewrites to follow their style). Refactorings and unit tests were also common large commits.

MySQL uses one repository per version. Its source code is a "fork". (Version 5.0 was forked from Version 4.1). Its use of Bitkeeper allowed a type of commit that we did not observed in any other project: changes in file permissions. MySQL programmers used a consistent lexicon and marked bug fixes with the word "fix" and merges with the word "merge". Merges were a very common type of large commit.

Firebird had many large build commits because of their support for multiple platforms. Their use of Microsoft Visual C++ project files often created as many build files as there were source files. These project files were also very prone to updating. Firebird had many large commits which were both module additions and cross cutting changes. Modules which updated or provided memory management, logging or various performance improvements were added and caused large cross cutting, far reaching changes. Since Firebird was an inherited code project from Borland, there were many large commits where old code was cleaned-up to follow the new code-style. A couple of large commits included the dropping of some legacy platform support.

Samba stores most of its documentation in the SCS and many of the documents were auto-generated from other documents; as a consquence many of the large commits were documentation driven. Samba had multiple branches developing in parallel, often new functionality was back-ported from develop branches to legacy branches. Thus merging was a common large commit for Samba. Large build and configuration commits included support for Debian builds, requiring a separate build module. Samba also went through a few security audits and the addition of cross cutting features such as secure string functions and alternative memory

Categories of Large Commits	Description
Implementation	New requirements
Maintenance	Maintenance activities.
Module Management	Changes related to the way the files are named and organized into modules.
Legal	Any change related to the license or authorship of the system.
Non-functional source-code changes	Changes to the source code that did not affect the functionality of the software, such as reformatting the code, removal of white-space, token renaming, classic refactoring, code cleanup (such as removing or adding block delimiters without affecting the functionality)
SCS Management	Changes required as a result of the manner the Source Control System is used by the software project, such as branching, importing, tagging, etc.
Meta-Program	Changes performed to files required by the software, but which are not source code, such as data files, documentation, and Makefiles.

Table 5: Categories of Large Commits. They reflect better the types of large commits we observed than those by Swanson.

management routines. Except for the Debian related commits, we were surprised to see very few platform support commits (Samba runs in almost any flavour of Unix).

Egroupware integrates various external PHP applications into one coherent unit. Egroupware consistently had a large number of template changes (in many cases the templates contained code). Many large commits were due to the importing of externally developed modules which would then had to be stylized and integrated into Egroupware. The integration of external modules also implied a lot of merging of updated external modules. There was also a surprising number of commits where only version strings were changed.

Enlightenment is known for fancy graphics and themed window decorations. "Visual" themes (a group of images, animations and source code defining a look) are the most notable part of the large Enlightenment commits. Large window manager themes are imported and updated regularly. Enlightenment uses separate directories to do versioning and merges are often copying or moving files from one directory to another. Enlightenment also attracted a lot of small widgets and tools that shared Enlightenment visual themes, like clocks and terminal emulators. Some large commits consisted of imports of externally developed tools into the repository. There were very few tests in Enlightenment.

Spring Framework had many large commits that consisted of examples and documentation. An example application, Pet Store, was added and removed multiple times. Branching in the Spring Framework often was done at the file level, with many large commits being moves from the sandbox directory to the main trunk. Much Spring Framework code was associated with XML files that configured and linked spring components. Other very common large commits were refactorings, renamings, and module and file moves. Spring required large cross cutting changes.

PostgreSQL was peculiar for its emphasis in cleaning up its source code. Many of its large commits were code reformatting and code cleanup (such as removing braces around one statement blocks–PostgreSQL is written in C). Many of its large commits involved a single feature (suggesting cross-cutting concerns). Many of these commits were external contributions (from non-core developers).

Evolution makes extensive use of branching: many of its large commits reflect the use of branches for different versions, and to develop and test new features before they are integrated into the main trunk of the version control system. It also uses many data files (such as time-zone information) that are usually updated at the same time. Evolution uses clone-by-owning: it makes a copy of the sub-project *libical* every time this project releases a new version.

3.2 Quantitative Analysis

3.2.1 Types of Changes

We first present the distribution for each of the types of changes. Figure 1 shows their proportion by project. Each bar corresponds to all changes for that type, and it is divided per project (for example, for maintenance changes, 60% are to Spring Framework, and the rest 40% to Evolution). Figure 2 groups them together, as a percentage of the total number of commits for all projects (maintenance changes are slightly less than one percent of all commits). They show a lot of variation, but the most frequent changes were Adding Modules, Documentation, Initializing Modules, Feature Additions, and Merges. Table 3 shows the corresponding percentage of commits for each type in descending order. When comparing both figures, it is interesting to see that those types of changes dominated by one or two applications (such as Data, Split Module, Debug, Maintenance) occur very rarely.

3.2.2 Extended Swanson Maintenance Categorization

Figure 3 shows, for each project, the distribution of changes according to the Extended Swanson Maintenance Categories (see table 2). Figure 4 shows the aggregation of all projects. All projects show commits in each category, although in some cases (such as Spring and Firebird) they were dominated by one or two categories.

Table 8 shows what percentage of large commits that belonged to each Extended Swanson category. We compared our results (ignoring non-functional and implementation changes) to Purushothaman et al.'s[9]. Purushothaman divided changes into: Corrective, Perfective, Adaptive, Inspections, and Unknown. They found that 33%, 8%, 47%, 9%, 2% of small changes fell into each of these categories. We found the following: adaptive changes were the most frequent in both large and small changes; Small changes, however, were more likely to be corrective than perfective. For large commits the trend is inverted: perfective changes

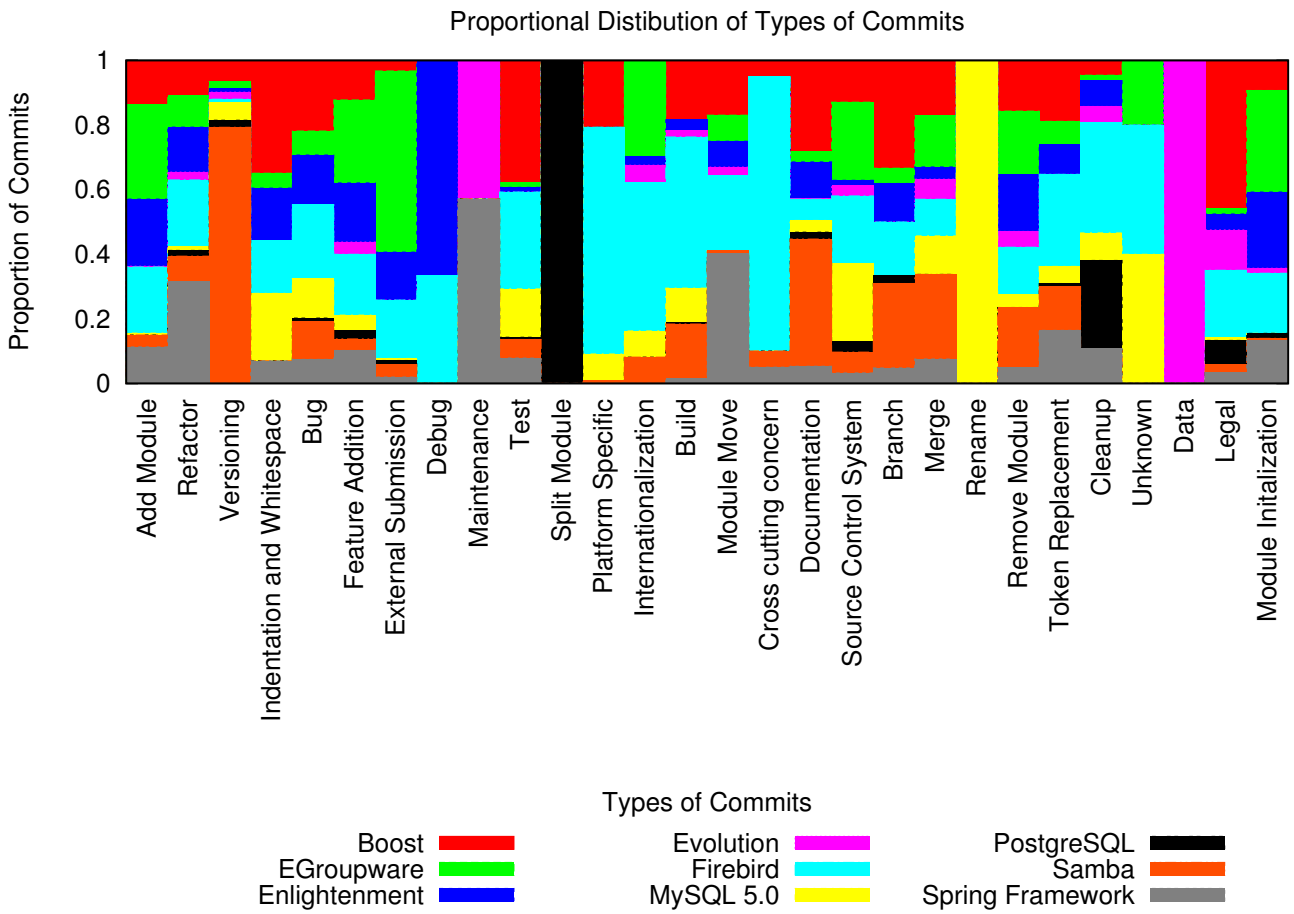

Figure 1: Distribution of Types of Changes by project. Each bar corresponds to 100% of commits of that type and it is divided proportionally according to their frequency in each of the projects.

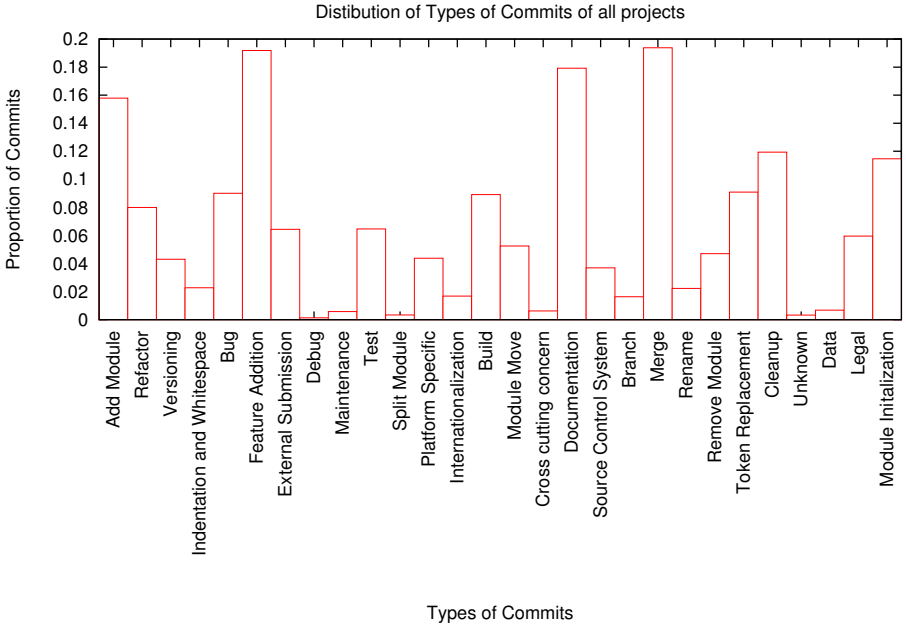

Figure 2: Distribution of Types of Change of the aggregated sum of all projects.

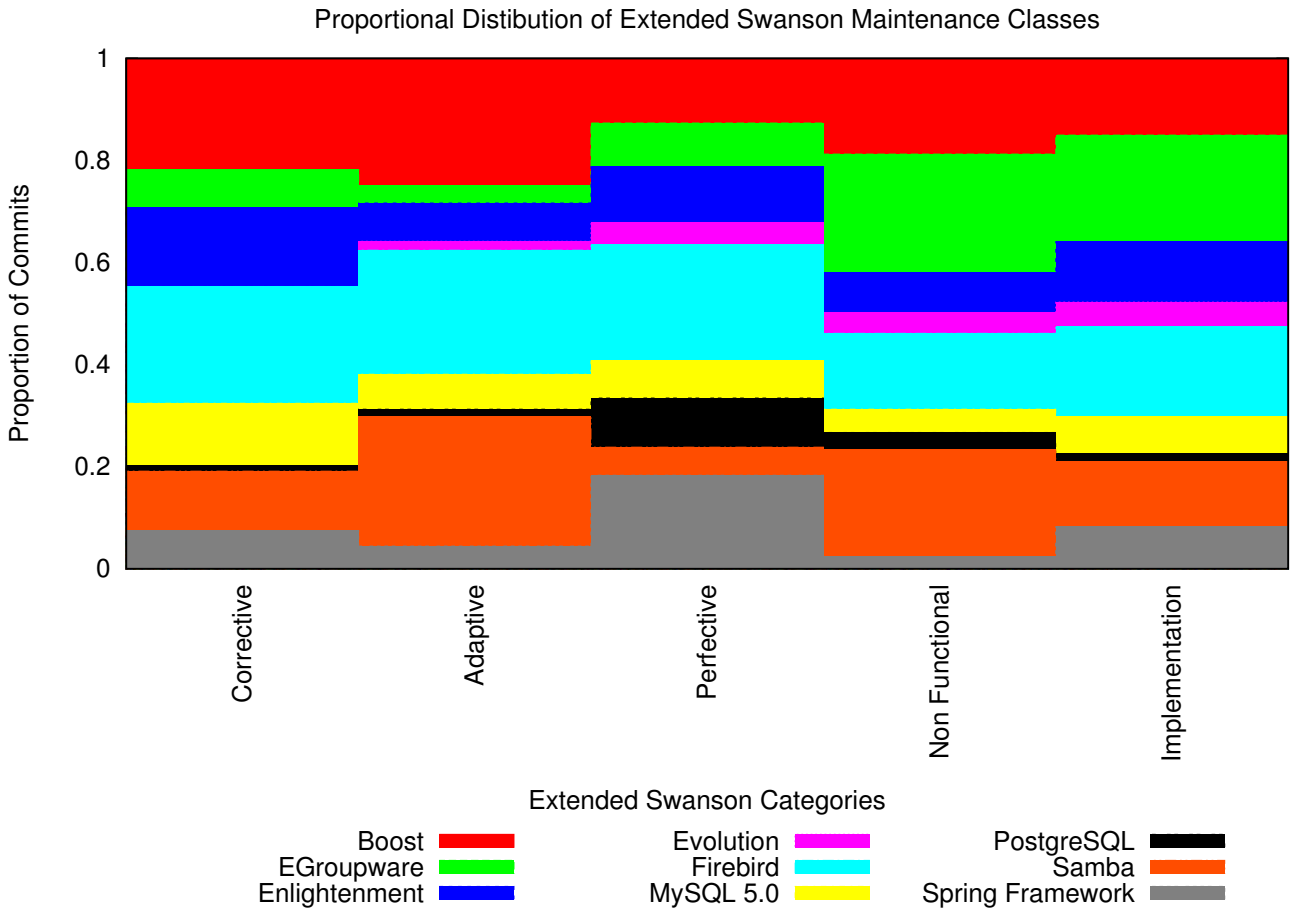

Figure 3: Distribution of changes per project, organized using Extended Swanson Maintenance Categories.

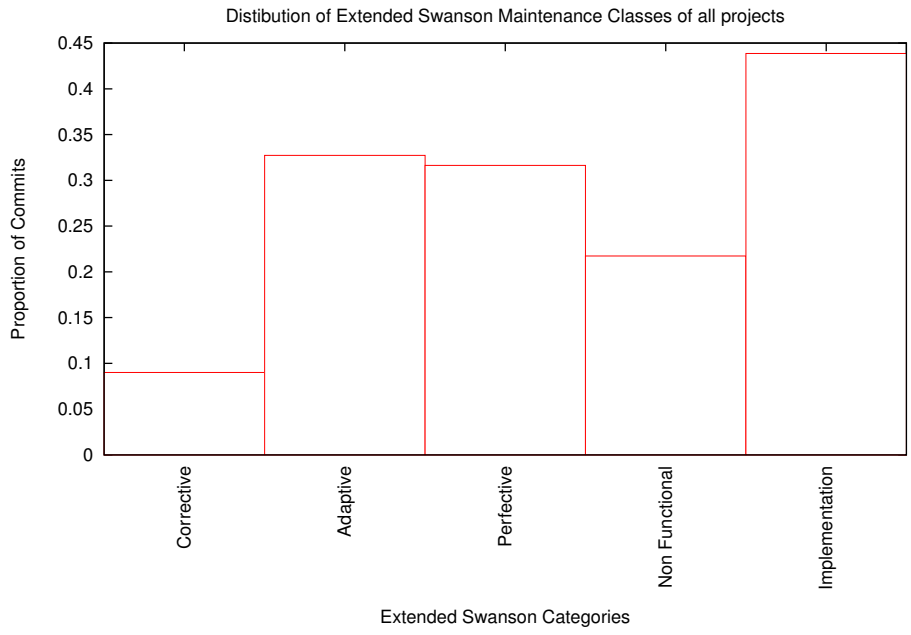

Figure 4: Distribution of changes for all projects, organized using Extended Swanson Maintenance Categories

Type of Change	Percent
Merge	19.4 %
Feature Addition	19.2 %
Documentation	17.9 %
Add Module	15.8 %
Cleanup	11.9 %
Module Initalization	11.5 %
Token Replacement	9.1 %
Bug	9.0 %
Build	8.9 %
Refactor	8.0 %
Test	6.5 %
External Submission	6.4 %
Legal	6.0 %
Module Move	5.3 %
Remove Module	4.7 %
Platform Specific	4.4 %
Versioning	4.3 %
Source Control System	3.7 %
Indentation and Whitespace	2.3 %
Rename	2.2 %
Internationalization	1.7 %
Branch	1.6 %
Data	0.7 %
Cross cutting concern	0.6 %
Maintenance	0.6 %
Split Module	0.3 %
Unknown	0.3 %
Debug	0.1 %

Table 7: Distribution of commits belonging to each Type of Change over all projects (1 commit might have multiple types).

Category	Percent
Implementation	43.9 %
Adaptive	32.7 %
Perfective	31.6 %
Non Functional	21.7 %
Corrective	9.0 %

Table 8: Proportion of changes belonging to each Extended Swanson Maintenance Category for all projects (1 commit might belong to multiple ones).

are more likely to occur than corrective changes. This result seems intuitive: correcting errors is often a surgical, small change; while perfective changes are larger in scope and likely to touch several files.

3.2.3 *Categorization of Large Commits*

Figure 5 shows the distribution of changes when they were classified according to our categories of large commits (see table 5). Projects vary in the distribution of the different categories. These distributions appear to support qualitative findings described in section 3.1; for example, Boost has the largest proportion of Meta-Program (mostly documentation) while Postgresql has the largest proportion of Non-functional Code (its frequent reformatting of the code).

Figure 6 shows the accumulated distribution of all the projects (summarized in table 9). The most common category is Implementation, followed closely by Meta-Program and Module Management. While some of the largest commits might be Legal, they represent the smallest category.

Category	Percent
Implementation	40.8 %
Meta-Program	31.5 %
Module Management	29.3 %
Non-functional Code	20.7 %
SCS Management	15.9 %
Maintenance	10.0 %
Legal	6.0 %

Table 9: The percent of commits belonging to each Large Maintenance Category over all projects (1 commit might belong to multiple categories).

4. ANALYSIS AND DISCUSSION

Large commits occur for many different reasons, and these reasons vary from project to project. They tend to reflect several different aspects:

Their development practices. Branching and merging results in large commits, but not all projects use it. Branching and merging is primarily used to provide a separate area for development, where a contributor can work without affecting others. Once the developer (and the rest of the team) is convinced that the code in the branch is functioning properly (e.g. the feature or features are completed) the branch is merged back to the trunk (the main development area of the repository). If a project uses branching and merging is because (we hypothesize) they prefer to develop and test a feature independently of the rest, and commit to the trunk a solid, well debugged large commit–instead of many, smaller ones. Branches become sandboxes. Evolution is an example of such project.

Their use (or lack) of the version control system. Many projects extensively use the features of a version control system, and this is reflected in the large commits (maintaining different branches for each version of the product–such as Evolution), while others don't (Enlightenment maintains different directories for different versions). Some projects use different repositories for each version (e.g. MySQL).

The importance of code readability. Some projects worry a lot about the way their code looks (Boost and PostgreSQL) and they do regular commits to make sure the code obeys their coding standards.

Externally produced features. Some projects accept a significant number of contributions from outsiders (to the core team) and these are usually committed in one single large commit (e.g. PostgreSQL, Egroupware).

Automatically generated files. Many large commits are the result of automatically generated files (Boost and Samba). This could be a concern for researchers creating tools that automatically analyze software repositories without properly determining the provenance of a given file.

Test cases and examples tend to be added in large commits. It is also interesting that not all projects have test suites nor examples (where appropriate).

Clone-by-owning results in regular large commits. Some projects keep a local copy of another product that they depend upon, such as a library, but they do not maintain it; such products need to be regularly updated with newer

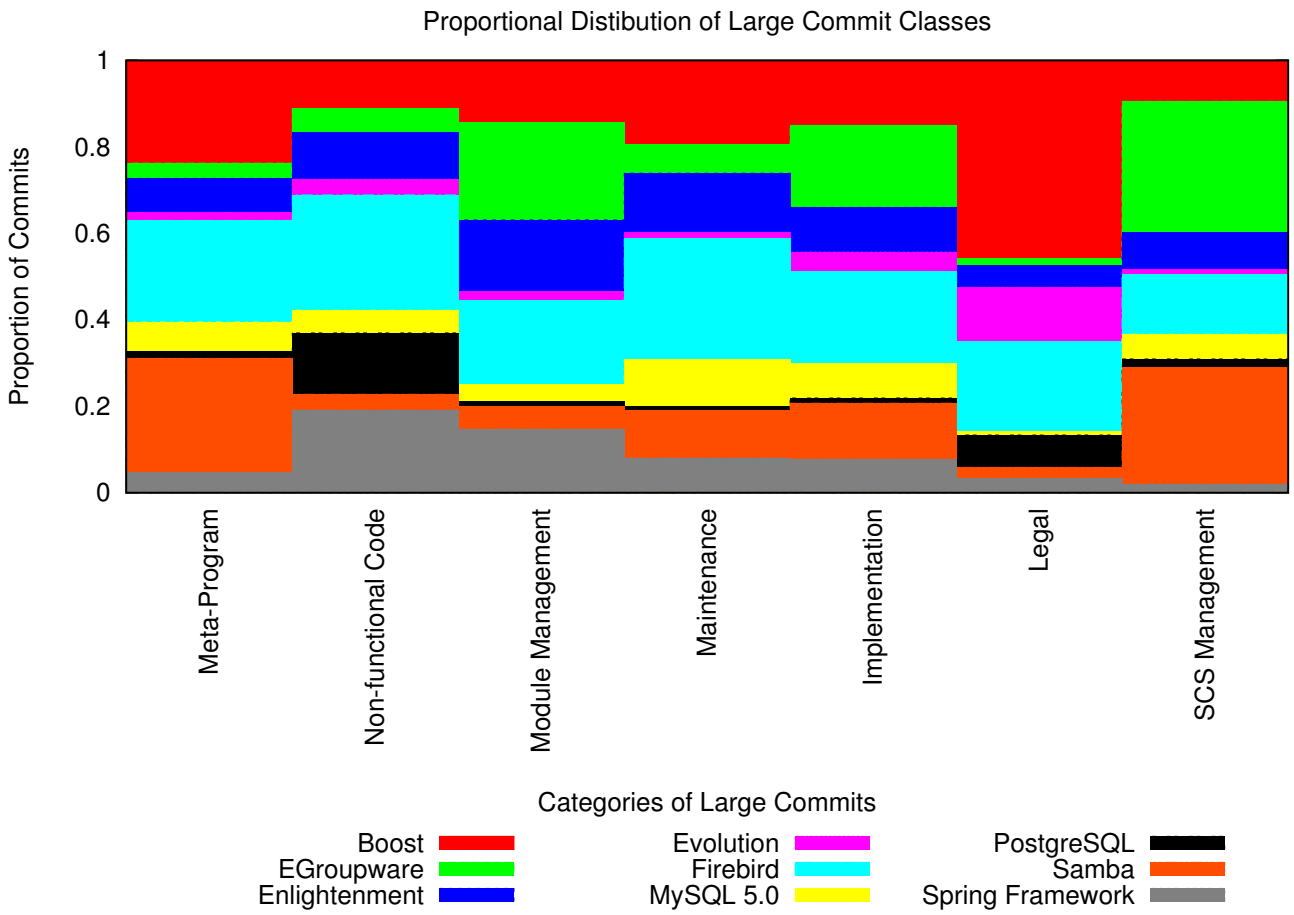

Figure 5: Distribution of commits per project, classified according to the Categories of Large Commits.

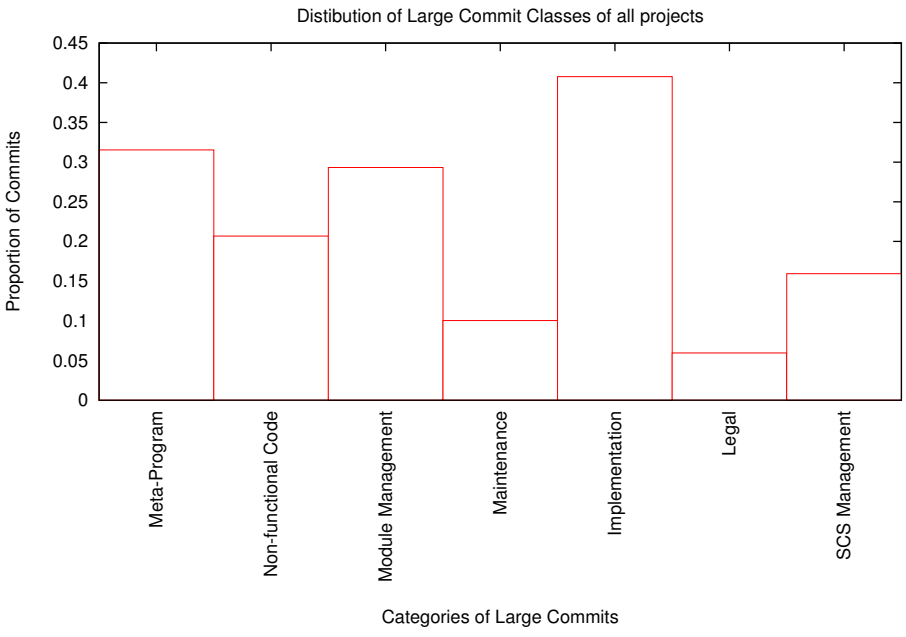

Figure 6: Distribution of commits for all projects, classified according to the Categories of Large Commits.

versions[2]. These updates are performed on a regular basis, and result in large commits. A project can look (to the outsider) as if it is having significant activity, when in reality is just copying code from somewhere else.

The copyright and licensing changes can be useful in tracking the legal provenance of a product. For example, who have been their copyright owners, or its changes in license (e.g. Evolution has had many different copyright owners during its lifetime, Boost changed license).

The development toolkit has an impact in large commits. We noticed that the use of Visual Studio prompted significantly more large commits to build files, compared to cmake or autoconf/automake (used by many projects).

The impact of the language. Large commits for languages such as C++ and Java contained a lot of refactorings and token replacement. This might be not a result of the feature of the language, but the manner in which programmers of that language tend to work (refactoring is perhaps more likely to be performed by programmers of object oriented languages; it is also possible that C++ and Java programmers were more likely to use the term "refactoring" in their commit logs, and this had an effect in the way we classified their commits). Spring, Firebird and Boost (written in Java and C++) contained many API changes and refactorings.

We believe that reading a commit log and its diff gives an idea of how easy or difficult it is to maintain a system. For example, we observed that several features in PostgreSQL required large commits to be implemented. This is very subjective, but reliable methods could be researched and developed to quantify such effect.

4.1 Threats to Validity

Our study unfortunately suffers many threats to validity. Our main threats were were consistency in annotation, subjectivity of annotation, and miscategorization. Much of the annotation relied on our judgement, as programmers, of what the commit probably meant. In the case of MySQL we did not use the source code, just the filenames and change comments. The classes were assigned by us and in many cases could be assigned subjectively. There were also only two people annotating the data, thus there might be bias and disparity in the annotation step.

Another issue is our choice of projects, we only chose nine relatively large OSS projects, do our results scale up and down? Do our results apply to non-open source products?

5. CONCLUSIONS

Although large commits might look like outliers in the large data-sets extracted from SCSs we have shown that in many cases these commits are fundamentally important to the software's very structure. Many of the large commits actually modify the architecture of the system.

We compared our study with another study of small changes and found an important difference between small changes and large commits was that large commits were more likely to perfective than corrective, while small changes were more often corrective rather than perfective. In a way it makes sense, correcting errors is surgical, perfecting a system is much more global in scope.

We have shown that the large commits provide insight into the manner in which projects are developed and reflect upon the software development practices of its authors.

Future work will exploit this data-set further. We plan to apply a lexical analysis of the diffs and change comments, in an attempt to automatically classify large commits. We would also like to analyze more projects to see if these results actually generalize more than what we have seen.

6. REFERENCES

[1] A. Capiluppi, P. Lago, and M. Morisio. Characteristics of Open Source Projects. In *CSMR '03: Proceedings of the Seventh European Conference on Software Maintenance and Reengineering*, page 317, Washington, DC, USA, 2003. IEEE Computer Society.

[2] D. M. German. Decentralized Open Source Global Software Development, the GNOME experience. *Journal of Software Process: Improvement and Practice*, 8(4):201–215, 2004.

[3] D. M. German. Mining CVS repositories, the softChange experience. In *1st International Workshop on Mining Software Repositories (MSR 2004)*, pages 17–21, May 2004.

[4] D. M. German. A study of the contributors of PostgreSQL. In *3rd International Workshop on Mining Software Repositories–MSR Challenge Reports (MSR 2006)*, May 2006. Received *Best Challenge Report* Award.

[5] D. M. German and A. Hindle. Measuring fine-grained change in software: towards modification-aware change metrics. In *Proceedings of 11th International Software Metrics Symposium (Metrics 2005)*, 2005.

[6] M. W. Godfrey and Q. Tu. Evolution in Open Source Software: A Case Study. In *Proceedings of International Conference on Software Maintenance*, pages 131–142, 2000.

[7] T. Mens and S. Demeyer. Evolution Metrics. In *IWPSE '01: Proceedings of the 4th International Workshop on Principles of Software Evolution*, New York, NY, USA, 2001. ACM Press.

[8] 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.

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

[10] 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.

[11] T. Zimmermann and P. Weisgerber. Preprocessing CVS data for fine-grained analysis. In *1st International Workshop on Mining Software Repositories*, May 2004.

[2]Clone-by-owning is usually done to avoid unexpected changes to the original software; it can also be used to simplify the building process.