2013-02-27

Cliff Walls: Threats to Validity in Empirical Studies of Open Source Forges

Landon James Pratt
Brigham Young University - Provo

Cliff Walls: Threats to Validity in Empirical Studies of Open Source Forges

Landon J. Pratt

A thesis submitted to the faculty of
Brigham Young University
in partial fulfillment of the requirements for the degree of

Master of Science

Charles D. Knutson, Chair
Daniel Zappala
Quinn Snell

Department of Computer Science

Brigham Young University

February 2013

ABSTRACT

Cliff Walls: Threats to Validity in Empirical Studies
of Open Source Forges

Landon J. Pratt
Department of Computer Science, BYU
Master of Science

Artifact-based research provides a mechanism whereby researchers may study the creation of software yet avoid many of the difficulties of direct observation and experimentation. Open source software forges are of great value to the software researcher, because they expose many of the artifacts of software development. However, many challenges affect the quality of artifact-based studies, especially those studies examining software evolution. This thesis addresses one of these threats: the presence of very large commits, which we refer to as "Cliff Walls." Cliff walls are a threat to studies of software evolution because they do not appear to represent incremental development. In this thesis we demonstrate the existence of cliff walls in open source software projects and discuss the threats they present. We also seek to identify key causes of these monolithic commits, and begin to explore ways that researchers can mitigate the threats of cliff walls.

Keywords: software engineering, open source software, cliff walls, software evolution, version control, repository mining, latent dirichlet allocation, SourceForge, Apache Foundation, artifact

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Artifact-Based Software Engineering Research

Artifact-based software engineering research may in some respects be compared to archaeology, a field that has been defined as "the study of the human past, through the material traces of it that have survived"[5]. Much like archaeologists, empirical software engineering researchers often seek to understand people. The software engineering researcher, while not separated from a target population by eons of time, faces other obstacles that often make direct observation impossible. Many organizations are loath to allow researchers through their gates, in an effort to protect trade secrets or merely to hide shortcomings. Even in cases where researchers are allowed to directly observe engineers building software, the Hawthorne effect[1] may threaten the validity of such observations. Time investment and organizational complexity also pose problems in software engineering research. Direct observation requires significant time investment, making it impossible for a single researcher to observe everything that takes place within a given software organization.

As a result of these barriers, software researchers, like their archaeological counterparts, take advantage of artifacts—work products left behind in software project burial grounds.

[1]The Hawthorne effect gets its name from the Hawthorne Works, a factory near Chicago where a series of experiments were conducted during the 1920s and 30s. The researchers in these experiments manipulated working conditions imposed on factory workers, attempting to measure the effect that these changes would have on productivity. In many cases the researchers found that productivity temporarily increased as the result of *any* change, even for changes that reverted a prior change (e.g., increasing lighting and then subsequently returning lighting to its previous level). This discovery was incidental to, and not tested as part of the experiments. The phenomenon later became known as the Hawthorne effect. One interpretation of the effect is that a change in working conditions was an indicator to participants that they were being observed, leading to increased effort on their part.

Artifacts are collected after the fact, minimizing the confounding influence of the presence of a researcher. Artifacts also help researchers deal with the requirements of studying complex organizations. By leveraging artifacts of the software process, researchers are able to study thousands of pieces of software in a relatively short period of time, an otherwise impossible task.

The open source software movement is increasingly important to software engineering research, since project artifacts, such as source code, revision control histories, and developer communications, are openly available to software archaeologists. Tens of thousands of open source projects are available on the Internet, found in software forges such as SourceForge and The Apache Foundation. This makes open source software an ideal target for researchers with a desire to understand how software is built.

1.2 Challenges of Artifact-Based Research

Unfortunately, the study of artifacts in software engineering is not all sunshine and double rainbows; serious challenges threaten the results of artifact-based research involving open source software projects. Since artifact data is examined separately from its original development context, identifying the development artifacts actually recorded in the data can be difficult. It is challenging enough to ensure that measurements taken for a specific purpose actually measure what they claim to measure. It is all the more difficult and necessary, therefore, to validate artifact data, which are generally collected without a targeted purpose. In the following paragraphs we provide an introduction to some of the threats to validity that open source researchers must address.

Many of the text-based files in open-source projects are not source code. Examples include documentation files, XML-based storage formats, and text-based data files such as maps for games. While they are not source code themselves, when included in a version control system, such as Subversion, changes to these files appear in much the same way. One study found that of the more than 20,000 unique file extensions identified in projects on

SourceForge, only 195 are considered "common source code extensions" [28]. Since many of the metrics used in software engineering research are targeted to source code, if researchers are not careful to filter out non-source-code files, attempts to measure aspects of software, such as project size and productivity, may be compromised.

Another danger in artifact-based research is the presence of auto-generated code. Unfortunately, in many projects auto-generated files are legitimate source code and don't exhibit any telltale characteristics that would assist researchers seeking to account for them. For example, Java user interface code is often generated by tools rather than written by hand. These files can be difficult to identify and almost certainly don't represent a unit of work analogous to manually written code.

Some version control systems support branching, a feature that enables concurrent development of parallel versions of a project. However, Zimmermann and Weißgerber note that merges can cause undesirable side-effects for two main reasons: 1) they group unrelated changes into one transaction and 2) they duplicate changes made in the branches [41]. One such side effect occurs when researchers attempt to estimate project size through analysis of CVS logs. Changes made in a branch are counted twice: first when they are introduced into the branch, and second when the branch is merged, resulting in a project size estimate inflated by as much as a factor of two. Merges can also falsely inflate measures of author contributions. All of the changes reflected in the merge transaction are attributed to the developer who performs the merge, regardless of whether or not that author actually produced any of those changes. If researchers do not take measures to correctly handle merges, analysis results may be unreliable.

These are just a few of the phenomena that jeopardize artifact-based research of open source software projects. Understanding the limitations of artifact data is integral to the agendas of several research communities (e.g., FLOSS, MSR, ICSE, and WoPDaSD) and is an important step toward validating the results of numerous studies (e.g., [6, 12, 18, 25, 31,

36, 40]). Despite on-going efforts to identify and mitigate the limitations of artifact-based research, new threats continue to emerge.

The focus of this thesis is one such threat—the presence of monolithic commits in open source repositories. A close look at the version control history of many projects in SourceForge reveals some worrisome anomalies. Massive commits, which we refer to as "cliff walls," appear with alarming frequency. One investigation of cliff walls in SourceForge found that of almost 10,000 projects studied, half contain a cliff wall representing 30% or more of the entire project size [28]. These cliff walls indicate periods of unexplained acceleration in development, threatening some common assumptions of artifact-based research, particularly for studies of project evolution. Cliff walls thwart attempts to tie authors to contributions, mask true development history, and taint studies of project evolution. We must better understand cliff walls if we are to paint an accurate picture of software evolution through the use of artifacts.

1.3 Related Work

1.3.1 Commit Taxonomies

The ability to distinguish between types of cliff walls is critical for many artifact-based studies. For example, a researcher attempting to measure developer productivity on a project would need to distinguish between large commits caused by auto-generated code and and those resulting from branch merges, as they must be handled differently when calculating code contributions.

Hindle, German, and Holt present a taxonomy used to manually categorize large commits, using commit messages, on a number of open source software projects [21]. Hattori and Lanza present a method for the automatic classification of commits of all sizes in open source software projects, also using commit messages [19]. The taxonomies developed in these studies focus on classifying the type of change made (for example, development vs. maintenance). Both of these studies conclude that corrective maintenance activities, such

as bug fixes, rarely yield large commits. These studies also find that a significant portion of large commits were dedicated to "implementation activities." The authors consider *any* source code file added to a repository to be the result of implementation activities. We feel that this is an oversimplification. For example, an external library added to the repository should not be considered part of an "implementation activity."

Arafat and Riehle categorize commits from open source projects into three groups, using a rough heuristic based on commit size [4]. Instead of analyzing commit log messages, they divide each commit into one of three groups, based on commit size. The authors note that there is much work still to be done in enhancing this heuristic.

While these studies begin to explore the phenomenon of large commits, they provide no sense as to whether monolithic commits threaten the results of artifact-based research of software projects. While we believe that cliff walls represent a threat to many aspects of artifact-based software engineering research, studies of software evolution are especially at risk.

1.3.2 Open Source Evolution

Software engineering researchers often seek to understand the evolution of software over time. Godfrey and Tu studied code growth over time within the Linux kernel [17]. Koch also examined code growth over time on a set of over 8,000 projects from SourceForge [24]. Nakakoji et al. examined evolutionary patterns of the communities and products for four open source projects: GNUWingnut, Linux Support, SRA-PostgreSQL, and Jun [32].

Mockus et al. examined email archives and version control history for the Apache web server and Mozilla browser open source projects and compared these with five commercial, closed source projects [31]. They considered some evolutionary variables, such as developer participation over time, in their study.

These few studies are just a small sample of studies of software evolution, the results of which are potentially confounded by the presence of very large commits. Unfortunately,

most researchers are largely unaware of this threat to validity, and the many others facing artifact-based studies.

1.3.3 Threats to Validity

Surprisingly little research addresses threats to validity in artifact-based research of open source software forges. Howison and Crowston identified a number of challenges in extracting and using data from SourceForge [22]. The authors introduced the concepts of "repository of record" and "repository of use," suggesting that some projects may use SourceForge only as a distribution mechanism. In these cases, the development history of the project would not be contained in the version control logs.

Ranier and Gale studied approximately 50,000 projects from SourceForge in an attempt to identify active projects [34]. This study looked at indicators of project activity, including number of commits, number of developers, and forum and mailing list activity. The authors concluded that fewer than 12% of the projects studied were in active development.

MacLean et al. introduced Cliff Walls, and discussed some of the challenges they pose to artifact-based software engineering research [27, 28]. Our proposed research builds on this foundation.

1.4 Thesis Statement

In this thesis we demonstrate the existence of very large commits, or "cliff walls," in open source software projects and discuss the threats that these present to many artifact-based studies, especially studies of software evolution. We also identify key causes of these monolithic commits, and begin to explore ways that researchers can mitigate the threats of cliff walls.

1.5 Project Description

One goal that we share with many software engineering researchers is to take advantage of the great quantity of software artifacts from open source projects. The main challenge

to artifact-based research in any field is ensuring that the artifacts mean what we think they mean. In order to accurately study software project evolution we need artifacts that represent *incremental development activities*. We first need some artifact representation of the events that occur as a part of software creation. Secondly, we need this representation to be sufficiently granular so that information loss is minimal. Thus we arrive at a conceptual definition of cliff walls—*an artifact representation of software development that is so large that one suspects that incremental development information has been lost.*

1.5.1 Operational Definitions

As we have explored cliff walls, we have developed several operational definitions for studying the phenomenon in practical settings. In our early treatment of cliff walls [27, 28], we examined projects from SourceForge using CVS. In CVS logs, the concept of a "commit" is not represented, as only individual file revisions are tracked. As a result, we decided to group revisions based only on the day in which they occurred. This aggregation became our representation of software development activity. We also did not specify a size threshold for cliff walls in these papers, instead examining the entire size distribution of daily code contributions. The results of our two initial studies of cliff walls are contained in Chapters 2 & 3 of this thesis. In a later study [33], in order to refine our treatment of cliff walls, we utilized the "sliding time window" approach introduced by Zimmermann to infer commits from CVS logs [41]. We believe the "commit" to be a better encapsulation of incremental development activities than that of revisions for a 24 hour period. Also, this same unit of measure is applicable to projects that utilize the Subversion VCS. We analyzed cliff walls in projects using Subversion in one study that is a part of this thesis [33]. Also, in our later studies, we specified a size threshold for identifying cliff walls—commits of size 10,000 LOC or greater were considered to be cliff wall commits. We believe this is a reasonable threshold, since a commit of this magnitude is unlikely to result from "normal" incremental development. The results of these studies are included in Chapters 4 & 5 of this thesis.

1.5.2 Commit Behavior

In [28], we hypothesized a number of causes for cliff walls: auto-generated code, branch merges, code imports and off-line development.

A number of tools produce source code that would otherwise need to be created by hand. GUI editors and documentation management systems are examples of tools that, if used extensively, could produce cliff walls. The code created by these tools, although often indistinguishable from hand-written code, is fundamentally different. Code that is automatically generated by a tool seldom represents the same developer effort as a manually written code segment of the same size. As a result, researchers concerned with measuring productivity or other project attributes would need to control for these differences.

Branching and merging may also contribute to the creation of cliff walls. After a branch is created, all development activity in the branch appears in version control, as one would expect. This poses no problem. However, when the branch is merged back into the development trunk, all the previous activity on the branch is duplicated in version control in the form of one giant commit. At least for CVS and Subversion, this commit does not contain a record of the intermediate changes in the branch, nor does it "correctly" attribute each change to its respective author; the developer who performed the merge gets "credit" for all changes submitted by the merge. If a significant amount of activity has occurred in the branch, the merge will likely result in a cliff wall. While there are ways to deal with this problem, researchers must first be able to identify merge commits.

"Off-line development" is another potential cause of cliff walls. We use the term off-line development to refer to large quantities of code developed as part of a project, but for which we have no record. Off-line development may include code developed without the benefit of version control, or developed in a separate repository and subsequently added to the SourceForge CVS repository in a monolithic chunk. While many projects in SourceForge utilize the version control system provided by the website, it is possible that others use SourceForge not as a development tool, but rather as a mechanism for code distribution.

Instead of submitting code changes to SourceForge as they occur, these projects maintain their own external version control. Updates are then made to SourceForge only when the developers wished to make the code publicly available, possibly as new versions are completed. Howison and Crowston discussed this issue, coining the terms "repository of use" and "repository of record" to refer to the separate repositories utilized on such projects [22]. Projects that maintain separate repositories for development and distribution almost certainly produce cliff wall commits.

A "code import" occurs when a significant amount of independently developed code is added to a project repository. The primary difference between a code import and off-line development is that, with a code import, the code was not developed as a part of the project of interest. Thus, the development history of a code import is not relevant to the development history of the project. One example of a code import is the inclusion of an externally developed library in version control.

1.5.3 Commit Classification

Before we can fully resolve the challenges presented by cliff walls, we need to identify the true causes. Thus, classification of cliff walls was the first task to be undertaken as part of this thesis. In section 1.3, we discussed a number of taxonomies and methods for the classification of author contributions in version control systems. While each of these taxonomies is useful, they fall short when applied to the cliff walls problem.

In this thesis, we focus on classifying large commits by identifying the causes behind them. In order to identify these causes, we extracted information from the commit log messages provided by developers with each code contribution. These messages contain valuable information on the motivations for each code change. We found that these messages also prove valuable in identifying the primary reasons behind very large commits. To extract this information we employed Latent Dirichlet Allocation, a topic model commonly used in Text Mining. In the next section, we discuss this method and our motivations for selecting it.

1.5.4 Latent Dirichlet Allocation

Latent Dirichlet Allocation (LDA) is an unsupervised, hierarchical Bayesian topic model for analyzing natural language document content that clusters the terms in a corpus into topics [8]. In LDA, the model represents the assumption that documents are a mixture of these topics and that each topic is a probability distribution over words. In short, LDA allows us, without any knowledge engineering, to discover the topical content of a large collection of documents in an unsupervised fashion. Given a corpus of documents such as our commit log messages, inference techniques such as Gibbs sampling or Variational Bayes can be used to find a set of topics that occur in that corpus, along with a topic assignment to each word token in the corpus.

A common technique for visualizing the topics found using LDA is to list the top n most probable terms in each topic. For example, in one analysis of commit log messages we performed, we found a topic concerning bug correction. This topic consisted of the following terms:

fix bug fixes crash sf problems small quick closes blah deleting weapon decoder lost hang weapons delphi noted led

It is important to note that the terms in an LDA topic are ordered based on their importance to the topic. This means that as you move down, the terms are less related to the topic. In the example above the terms **fix**, **bug** and **crash** are more important to this particular topic than are the terms **weapons** and **delphi**.

Because LDA also assigns a topic to each individual token within a commit message, a single commit can, and usually does, contain many topics. This is one benefit of using LDA: each "document" may belong to multiple classes. Such is not the case with many of the supervised classification methods, which typically assign each document to a single class. In the case of commit log messages, allowing for multiple topics in a single message allows us to conduct a more fined-grained analysis.

The approach taken by LDA is an attractive alternative to other classification and clustering methods. Supervised classification methods require a training set of tagged data, for which the appropriate class for each instance has previously been specified. In many situations this tagged data does not exist, requiring the researcher to manually assign tags to some portion of the data. This tedious, time-consuming, and potentially biased process can be avoided through the use of unsupervised methods such as LDA. Unsupervised methods may also compensate to some degree for short-sightedness on the part of the researcher. In supervised methods, since the classes must be predefined by the researcher, it is possible to degrade the usefulness of the model through the omission of important classes, or the inclusion of irrelevant classes. Unsupervised methods avoid some of these errors since no predefined clusters are specified, allowing the data to better "speak for itself." LDA was applied to commit log messages in the studies included as Chapters 4 & 5 of this thesis.

1.5.5 Solution

Simply identifying the causes behind cliff walls is not enough. Our ultimate goal is to improve the reliability of artifact-based software engineering research. In order to do this, we ideally would address each class of Cliff Wall commits, and understand how it impacts the study of projects in which it is present. In [28], we not only suggested causes for cliff walls, but we also identified potential solutions for each cause. The classification of cliff walls will allow us to know if the proposed solutions are appropriate for dealing with cliff walls.

We found that certain classes of cliff walls do not pose a significant threat to study results and can simply be ignored or removed with no negative effect. In some instances, however, cliff walls present a greater dilemma, forcing researchers to select projects more intelligently to avoid cliff walls. In these cases, the ability to filter out the projects most affected by cliff walls greatly improves study results. The methods used to classify and quantify the causes of Cliff Walls is instrumental in developing filter criteria. In this thesis

we establish the foundation; however, we leave much of the development of these criteria as future work.

SourceForge is one of many environments in which open source software is developed. There are various open source forges and foundations, each with its own tools, communities, policies, and practices that influence the software development that occurs therein. It is likely that some of these environments will prove to be more welcoming to those interested in studying the evolution of software than will others. As part of this thesis we investigated cliff walls within two forges: SourceForge and The Apache Foundation. This is a first step towards determining whether certain open source communities are better candidates for artifact-based research than others.

In order to analyze cliff walls in Apache Foundation projects, it was necessary to convert the Apache Foundation subversion logs into a format that could be consumed by Mallet, the tool used for the LDA analysis. As a result, we created a tool that converts Subversion logs into a database format compatible with Mallet.

1.6 Validation

In order to validate the results of this thesis, a number of questions needed to be addressed. First, it was necessary to demonstrate definitively that cliff walls exist. In [28] we found that half the projects in our SourceForge data set included a cliff wall constituting nearly a third of the project size. In another study, we found over 10,000 individual commits of size 10,000 LOC or greater from a sample of 9,999 projects. This yields an average of around 1 cliff wall per project [33].

Second, it was necessary to demonstrate that cliff walls pose a problem. Because of their size and frequency, cliff walls are a significant threat to many studies. In [27] we concluded that the results of one previous study [26] needed to be re-evaluated in light of the cliff walls phenomenon. We expect this to also be true for many other artifact-based studies.

1.7 Overview of Findings

In the following sections we briefly describe each of the papers included in this thesis. Note that because the chapters of this thesis are published as a series of independent papers, some background information is repeated in multiple chapters.

1.7.1 Threats to Validity in Analysis of Language Fragmentation on SourceForge Data

In this study we introduced the phenomenon of cliff walls, and discuss some potential causes of these abnormally large commits (namely, auto-generated files, internal development, and development pushes). The initial definition of cliff walls, introduced in this paper, was based on percentage of project size, not on the absolute size of the code contribution. This paper was published and presented at the 1st International Workshop on Replication in Empirical Software Engineering Research [27]. The full version of this paper is included as Chapter 2 of this thesis.

1.7.2 Trends That Affect Temporal Analysis Using SourceForge Data

In this study we hypothesized additional cliff wall causes—project imports, branching and merging—and suggested methods researchers may employ to mitigate their effects on statistical analysis. In this study, we again used a primitive definition of cliff walls, based on the percentage of project size. We found that "half of the projects in our data set have a cliff wall that is nearly a third of the project size." In other words, the contributions of a single day constituted a third or more of the total project size. This paper was published and presented at the 5th International Workshop on Public Data about Software Development [28]. The full version of this paper is included as Chapter 3 of this thesis.

1.7.3 Cliff Walls: an Analysis of Monolithic Commits Using Latent Dirichlet Allocation

In this study, we used Latent Dirichlet Allocation to extract topics from over 2 million commit log messages, taken from 10,000 SourceForge projects. The topics generated through this method were then analyzed to determine the causes of over 9,000 of the largest commits. We found that branch merges, code imports, and auto-generated documentation were significant causes of large commits. We also found that corrective maintenance tasks, such as bug fixes, did not play a significant role in the creation of large commits. This paper was published and presented at the 7th International Conference on Open Source Systems [33]. The full version of this paper is included as Chapter 4 of this thesis.

1.7.4 A Topical Comparison of Monolithic Commits in SourceForge and Apache Foundation Software Projects

In this study we examined the impact of cliff walls in two open source forges: SourceForge and Apache Foundation. We found that while cliff walls appear much more frequently in Apache Foundation than in SourceForge, the opposite is true when only the most problematic cliff walls are considered. Cliff wall commits occur in both SourceForge and Apache Foundation projects, and thus must be addressed by researchers conducting artifact-based studies on projects from these forges. The Apache Foundation is preferable to SourceForge for certain types of artifact-based studies (particularly those involving software evolution) since the most problematic types of cliff walls are less frequent there. This paper is ready for submission, but has not yet been published. The full version of this paper is included as Chapter 5 of this thesis.

1.8 Future Work

1.8.1 Diversity of Open Source Software Projects

Open source software projects reflect tremendous diversity. Most open source software is developed within software forges, which are themselves very diverse. Projects in a forge may congregate around a specific language or technology, as is the case with RubyForge and GitHub. Some forges are focused on developing strong communities around each project, while others seek only to provide tools for open source development. Open source forges even have different rules for participation—while anyone can create a project on SourceForge, the Apache and Eclipse foundations have processes for allowing new projects into their respective forges.

In this thesis, we analyzed monolithic commits in a very limited setting, examining cliff walls in CVS repositories from SourceForge projects and SVN repositories from Apache Foundation projects. While this subset of forges and repository technologies are fundamental parts of the Open Source ecosystem, there are vast numbers of open source projects that use different technologies and are found in other forges. As a result, our efforts cannot possibly tell the entire story of cliff wall commits.

There is a great opportunity for further work examining the impact of cliff walls in different settings.

1.8.2 Automatic Classification of Cliff Walls

We are pleased with the insight that LDA has provided concerning the cliff wall phenomenon—we now have a better idea of the actions that typically result in the creation of cliff walls. While we are able to identify global trends for cliff walls, we do not feel that our current methods are yet sufficient for accomplishing our ultimate goal—automated identification and resolution of individual cliff walls. In order to achieve these goals, we must be able to classify and handle individual cliff walls with greater confidence.

As future work towards this goal, we propose an approach more dependent on machine learning techniques. We feel that a supervised approach, with manually classified cliff walls for model training and test, would allow us to identify causes for individual cliff walls with greater accuracy. LDA provides a rich set of features that could be used to train an automatic classifier: per commit topic proportions, number of topics per commit, and even topic dyads or triads. In addition to LDA features, a supervised machine learning approach could take advantage of other information gathered from the version control systems themselves. For example, file path information and change type, where available, would provide extremely valuable data for an automatic cliff wall classifier. These sources of information may also prove useful in situations where LDA is unable to provide insight, such as in the case of blank or uninformative log messages.

Our hope is that such a classifier would also better distinguish between external code and project code imports (an extremely important distinction in determining how to properly handle cliff walls). We are highly optimistic that future investment in this area could one day lead to automated tools capable of measuring and improving the quality of software artifacts gathered from version control systems.

Chapter 2

Threats to Validity in Analysis of Language Fragmentation on SourceForge Data

Reaching general conclusions through analysis of SourceForge data is difficult and error prone. Several factors conspire to produce data that is sparse, biased, masked, and ambiguous. We explore these factors and the negative effect that they had on the results of "Impact of Programming Language Fragmentation on Developer Productivity: a SourceForge Empirical Study." In addition, we question the validity of evolutionary or temporal analysis of development practices based on this data.

2.1 Introduction

The present work began as a replication of a study by Krein, MacLean, Delorey, Knutson, and Eggett, "Impact of Programming Language Fragmentation on Developer Productivity: a SourceForge Empirical Study" [26]. This study explored the contributions of individual software developers in light of their use of multiple programming languages.

As authors of the original study, we desired to conduct a differentiated replication in order to explore the original question from another angle—that of project evolution. We hoped to clarify relationships between the nature of project growth and the language fragmentation of individual authors contributing to such projects. We expected that such a replication study would shed light on the potential impact of writing software in only one language, as opposed to developing in two or more languages.

In order to assess the impact of language fragmentation on project growth, we first needed to develop a technique for reliably measuring project growth in the context of our data set, which required understanding the growth patterns of projects on SourceForge. This intermediate objective yielded unexpected insights into the nature and usability of project data on SourceForge, particularly as it relates to the analysis of project evolution.

In this paper we present potential threats to validity, and insights into the limitations of SourceForge project data for understanding project evolution. In particular, we present our results in light of the replication we conducted to explore the effects on projects of developing software in multiple languages. In order to provide context for our insights, we provide background information on the original study which we sought to replicate. In addition to potential pitfalls inherent in SourceForge project data, we also describe threats to validity inherent in the study of language fragmentation and individual developer productivity using SourceForge data.

2.1.1 SourceForge as a Data Source

Over 100 researchers use the SourceForge Research Data Archive (SRDA)[38] hosted at Notre Dame to analyze development and distribution on SourceForge.net. Many utilize the data to study development behavior in open source projects[1].

Howison and Crowston enumerate several pitfalls in using SourceForge as a data source for research [22]. In section 2.2 we discuss our insights into the limitations of SourceForge for studying certain project attributes.

2.1.2 Language Fragmentation

The targeted study explored the correlation between language fragmentation and programmer productivity. Fragmentation is measured by language entropy (originally defined in [25]), with productivity defined as the number of lines of code committed to all "Production/Stable"

[1]In addition to the intrinsic value of understanding open source development, some argue that OSS is sufficiently analogous to commercial software that global conclusions are justified.

or "Maintenance" phase projects on SourceForge in a given month. In order to provide context for our discussion of threats to validity, we briefly present the definitional premises of the original study.

2.1.3 Data Set

For this study and [26] the data set (*Sample*) comprises all projects designated "Production/Stable" or "Maintenance" as of the SRDA dump of SourceForge. Project history was gathered from project inception through October 2006. The data was originally collected for [14].

2.1.4 Definitions

In this study we define two terms precisely.

Definition 1 Daily Commit *is the unit of work, defined as the net contributions, in lines of code, for all authors to a project on a given day.*

Although monthly contributions were the unit of work in the Fragmentation study, we found that *Daily Commit* unmasks certain attributes of the data that are not visible at a coarser granularity.

Definition 2 Project Size *is the total size of the project, in lines of code, at the most recent commit in the* Sample.

2.2 Project Attribute Pitfalls

Several attributes of SourceForge projects may lead to erroneous results if not properly considered: 1) Unnatural Project Growth Patterns (which we refer to as *Cliff Walls*), 2) Auto-Generated Source Code, 3) Internal Development, 4) Development Pushes, and 5) Small Projects. To address the first four items we initially examine the Java eXPerience FrameWork (JXPFW) project as an example of problems in the data that lead to erroneous or artificially

inflated results. After articulating the problems in the single case, we show that the same problems exist in a large percentage of the projects on SourceForge. Finally, we address the issues that arise when analyzing small projects.

2.2.1 Java eXPerience FrameWork

JXPFW is a moderately sized, "Production / Stable" project written primarily in Java. Other languages utilized in this project include XML, CSS, HTML, and XHTML. The project was chosen from 25 randomly selected projects because it had the largest *Daily Commit*.

As of August 6, 2006 JXPFW contained 160,946 total lines—placing it in the top quartile of all projects in the *Sample*—of which 63,720 were classified as source code[2]. At least 67,023 lines appear to be auto-generated files (discussed later).

2.2.2 Language Entropy

In order to explain the problems associated with using SourceForge data to analyze Language Fragmentation, we must first define entropy and its calcualtion. Entropy, defined as

$$E(S) = -\sum_{i=1}^{c}(p_i \times \log_2 p_i) \tag{2.1}$$

measures the "evenness" and "richness" of a distribution (p) of classes (c) in a system (S). In Language Entropy "evenness" describes how evenly a developer contributes code in multiple languages. For example, if a developer commited 100 lines of Python and 100 lines of Java in one month, he or she would have maximize entropy for two languages: 1.0. For 150 lines of Python and 50 lines of Java Language Entropy would be 0.811. "Richness" describes the number of languages employed by an author in a given month. Maximum entropy is

$$\log_2 c \tag{2.2}$$

[2]Files were classified as source code or not source code based upon file extension. Our list of extensions covers 99% of the data in the *Sample*.

where c is the number of languages (classes).

The authors found a negative correlation between language entropy and programmer productivity. However, further analysis casts doubt on the generality of these conclusions.

2.2.3 Cliff Walls

Abnormal growth spikes in a project (*Cliff Walls*) constitute a serious threat to validity in evolutionary research utilizing SourceForge project data. These cliff walls represent periods of time during which data for the project is masked, missing, or ambiguous. Figure 2.1 shows the growth of the Java eXPerience FrameWork over time. The project was created on September 8, 1999 but sat dormant for over two and a half years before any source code was committed.

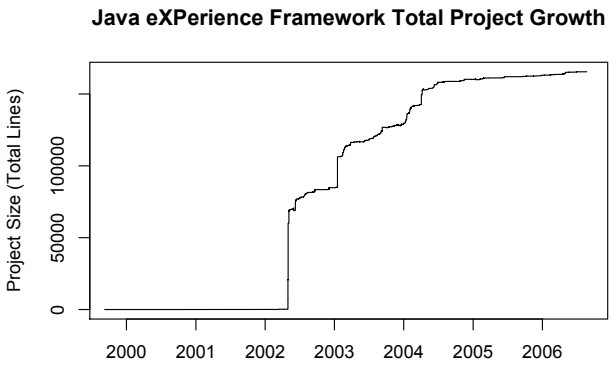


Figure 2.1: Growth of the Java eXPerience FrameWork over time.

On May 1, 2002, 138 new source code files were added to the project by a single author, totaling 18,675 lines of code (see Figure 2.2). In addition, another 62 lines were modified in 9 existing files. Auto-generated source code, internal development, gate-keepers, and development pushes are all developmental practices that could lead to these abnormally large commit sizes. We discuss each of these in turn in the following sections.

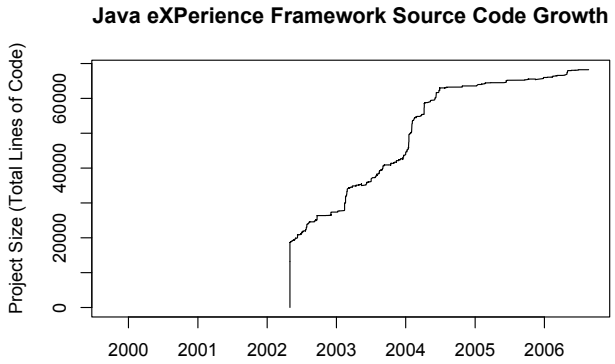


Figure 2.2: Growth of the Java eXPerience FrameWork over time.

2.2.4 Auto-Generated Files

42% of the lines in JXPFW are auto-generated .mdl files marked as binary in CVS. However, unlike image files such as .gif, .mdl line count is included in the *lines_added* statistic. In JXPFW these files are easy to identify due to their extreme size in proportion to the *Project Size* and the fact that they are marked binary. Unfortunately, in many projects auto-generated files are legitimate source code and don't exhibit any telltale characteristics. For example, Java user interface code is often generated by tools rather than written by hand. These files can be difficult to identify but probably don't represent a unit of work analogous to code written by hand.

No correction is made in the Language Fragmentation paper for auto-generated files, although the issue is listed in the threats to validity. These commits can significantly alter the result of the language entropy calculation. If auto-generated code is committed in a language that otherwise represents a small proportion of an author's efforts (such as auto-generated XML configuration files), language entropy is artificially increased. Conversely, if the auto-generated code is in a dominant language (such as Java user interface code), language entropy is artificially decreased. Both results cast doubt on the validity of the original calculation.

2.2.5 Internal Development

Another possible cause of *Cliff Walls* is internal development not yet stored on SourceForge. Such activity may result from corporate sponsorship or co-located developers who find it easier to collaborate locally. In both cases, SourceForge essentially becomes a distribution tool rather than a collaboration environment. In fact, 12.2% of the projects were only active on a single day (1,221 of 9,997). In addition, 50% of the projects had fewer than 17 active days (5,004 of 9,997). One project, "ipfilter" was active for a two and a half hour period on August 6, 2006, in which 71,878 lines were checked in. No changes were made before the data was extracted four months later.

Additionally, projects may experience periods of internal development. For example, shortly before release commits may be restricted to only allow bug fixes. These intermittent stages of restrictive commits may cause periodic cliff walls.

When development occurs outside of SourceForge, the data committed to the public repository is of such coarse granularity that conclusions about development efforts and practices based on the revision data are suspect. In Figure 2.2 we see that there appears to be no development activity for the first two and a half years of the project. However, given the large commit size on May 1, 2002, it is probable that the growth approximates Figure 2.3. Unfortunately, without further data we can only make educated guesses.

2.2.6 Development Pushes

Although auto-generated files and internal development may be the culprits in some cases, in other cases large commits may simply indicate an impending deadline or "development push." While it is unlikely that all of the developers on a project wrote 20% of a project in a single weekend, it is not impossible, and therefore cannot be discounted. Distinguishing between development pushes and artificially inflated commits is extremely difficult, and requires the acquisition of knowledge about a project from non-code sources such as email lists, bug reports, and interviews.

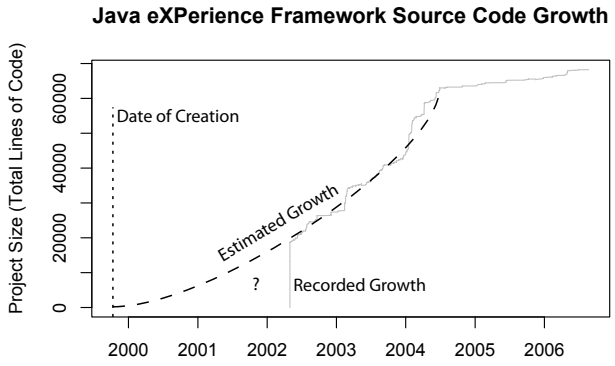


Figure 2.3: Growth of the Java eXPerience FrameWork over time with estimate of actual growth.

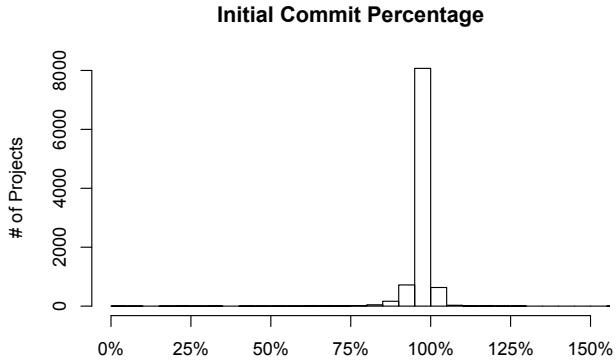
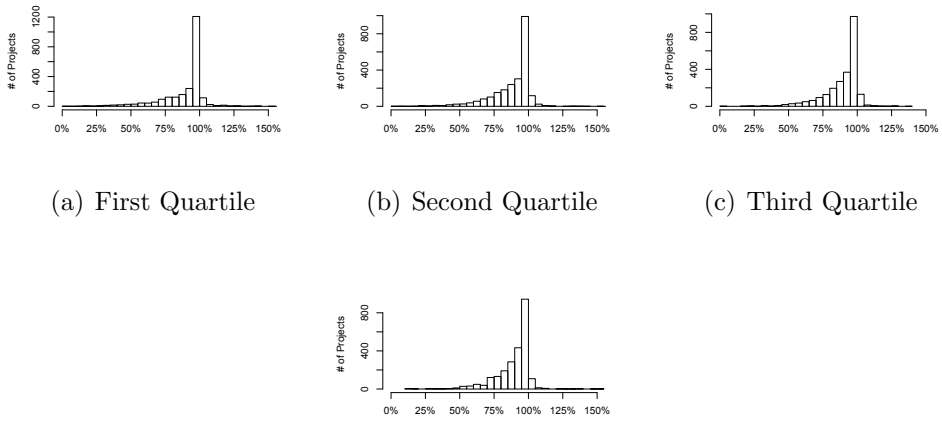


Figure 2.4: Percentage of the *Project Size* explained by initial commits.

2.2.7 Generalizing Pitfalls

The *Cliff Walls* demonstrated in JXPFW occur frequently in the *Sample*. Over 4,000 projects are made up almost entirely of initial commits, meaning that the files were checked in at their maximum size (see Figure 2.4). This effect is most pronounced in projects whose size lies in the first quartile[3], and is only slightly less pronounced in the other three (see Figure 3.6).

[3]Project size quartiles are: 1) 2 to 3,661.25, 2) 3,661.25 to 15,027, 3) 15,027 to 54,688.25, and 4) 54,688.25 to 27,283,364



(a) First Quartile (b) Second Quartile (c) Third Quartile

(d) Fourth Quartile

Figure 2.5: Percentage of the *Project Size* explained by initial commits by *Project Size* quartile.

2.2.8 Small Projects

The simplest explanation for cliff walls in the data is small project size. If a 4,000 line *Daily Commit* is made to a project with a size of 8,000 lines, that commit represents 50%[4] of the *Project Size*. However, the same *Daily Commit* to a project with a *Project Size* of 500,000 lines is negligible. Given the analytical impact of this phenomenon, we explore a possible explanation for small projects in the *Sample*, given the requirement that the projects are marked "Production/Stable" or "Maintenance."

78% (3,197 of 4,094) of projects with *Project Size* less than 10,000 and 57% (5,688 of 9,997) of all projects in our *Sample* have only a single author. Projects that can be developed and maintained by a single author are generally smaller than those developed and maintained by a dozen or more authors. To complicate matters, a single author has no need for collaborative tools, and may therefore be less likely to "commit early, commit often." Instead, small projects often exhibit peculiarities unique to an author and not relevant in discussions of collaborative product development.

[4]The dramatic cliff wall in Figure 2.1 is just shy of 50% of the total project size

2.3 Author Behavior Pitfalls

In addition to the pitfalls of project data that we discuss in the previous section, we observe a number of problems with author data that render analysis difficult or problematic. Several limitations are identified in the original study that we set about to replicate [26]. In light of the *Cliff Walls* and other limitations with projects, in this section we address Marginally Active Developers and Non-Contributing Months. Finally, we explore Author Project Size Bridging as an additional limitation.

2.3.1 Marginally Active Developers

Krein, et. al. [26] state that marginally active developers—those who contribute code during a limited number of months—may bias the results since they may be less likely to write in multiple languages. Observed at a granularity of one month, these authors represented few data points and therefore were less likely to generate realistic regression lines in the random coefficients model. We suggest addressing potential errors in analysis (discussed in Section 2.4) as well as using a finer granularity to mitigate the effects of low productivity. If developers are truly developing in multiple languages concurrently, this approach would reveal that limitation while still capturing language entropy.

2.3.2 Non-Contributing Months

In addition to marginal activity, many developers don't contribute regularly. Krein, et. al., recognize the potential problems with this data masking, and filter the data to remove abnormally large commits. However, given the *Cliff Walls* exposed in Section 2.2, this approach is likely insufficient to mitigate the potentially induced error.

Figure 2.3 shows a time period of two and a half years during which development occurred, but for which data is entirely missing. In [26] the entire development period would have been analyzed as the contributions for a single author (named *keess*) in the month of May, 2002. Classes for language entropy for the month include HTML, Java, XML, SQL, and

CSS. Although this data point for JXPFW would have been excluded in [26] due to its size (19,881 lines of code), it is easy to imagine a similarly drawn-out development process with a smaller total size that fell below the threshold of exclusion. Alternatively, if four developers had collaborated on the JXPFW commit, it would not have been filtered.

Regardless of the size of *Cliff Walls* following months of inactivity, the potential for errors in the language entropy calculation is high. Researchers must take care to filter or categorize anomalous commits to accurately analyze developer activity.

2.3.3 Author Project Size Bridging

Analysis of author contributions reveals that authors don't often bridge between project sizes (see Table 2.1). Authors tend to contribute to projects of a similar size and tend not to cross project size boundaries. To illustrate, we first must discretize projects into groups. Figure 2.6 shows a discretization by quartile on contributions ordered by project size[5]. In other words, 25% of the contributions were to projects of size 0 to 102,852, 25% to projects of size 102,852 to 337,858, etc.

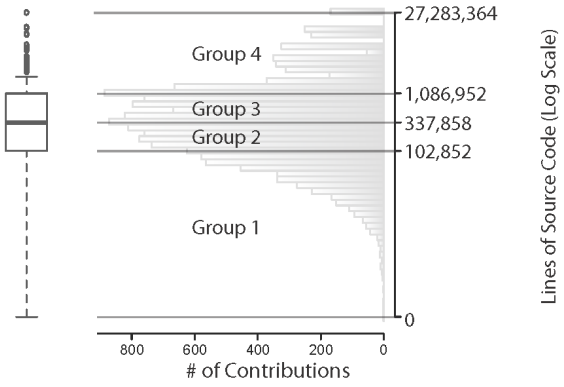


Figure 2.6: Project size groups.

If projects are discretized into groups using these quartiles, only 6.82% (1,499 of 22,095) of authors contribute to projects in multiple groups. 93.18% of authors contribute

[5]Because y is on the log scale, the bins towards the top of the histogram cover much greater range than the bins towards the bottom (notice the range of Group 1, 102,852, compared to the range of Group 4, 26,196,412). To elucidate this, we provide a boxplot. The boxplot demonstrates that most of the data in Group 4 are outliers.

	Group		Combined	
Group	Authors	Percentage	Authors	Percentage
1	11,632	52.90%	20,491	93.18%
2	4,202	19.11%		
3	3,024	13.75%		
4	1,633	7.43%		
1, 2	633	2.88%	804	3.66%
2, 3	133	0.51%		
3, 4	58	0.26%		
1, 3	340	1.55%	577	2.53%
2, 4	151	0.69%		
1, 4	66	0.30%		
1, 2, 3	79	0.36%	87	0.40%
2, 3, 4	8	0.04%		
1, 2, 4	25	0.11%	42	0.19%
1, 3, 4	17	0.08%		
1, 2, 3, 4	9	0.04%	9	0.04%
	21,990	100.00%	21,990	100.00%

Table 2.1: Author bridging, or lack thereof, between *Project Size* groups (see Section 2.3.3).

within a single contribution group. Another 3.66% bridge only between contiguous groups. This general lack of bridging suggests that the author data represent four distinct populations, rather than one. If true, this assertion requires that researchers block analysis of author productivity and contribution by contribution group.

When the same analysis is performed on only those authors who contribute to multiple projects, the contrast is not as extreme (see Table 2.2). However, only 13.14% (2,891 of 21,990) contribute to more than one project. Since removing single project authors also strips out nearly 90% of the data, doing so is not a viable solution.

Further analysis of author contribution is required to determine the meaning of the 5.15% of authors who bridge between contiguous groups. We expect that shifting the boundaries of the groups will increase the number of authors who contribute to a single group.

Group	Group Authors	Group Percentage	Combined Authors	Combined Percentage
1	1,197	41.4%		
2	75	2.59%	1,392	48.15%
3	112	3.87%		
4	8	0.28%		
1, 2	633	21.90%		
2, 3	133	3.91%	804	27.81%
3, 4	58	2.01%		
1, 3	340	11.76%		
2, 4	151	5.22%	577	19.27%
1, 4	66	2.28%		
1, 2, 3	79	2.73%	87	3.01%
2, 3, 4	8	0.28%		
1, 2, 4	25	0.86%	42	1.45%
1, 3, 4	17	0.59%		
1, 2, 3, 4	9	0.31%	9	0.31%
	2,891	100.00%	2,891	100.00%

Table 2.2: Author bridging, or lack thereof, between *Project Size* groups for authors who contribute to multiple projects.

2.4 Limitations in the Original Study

In [26], the authors use *lines_added* as the metric for productivity. They argue that *lines_added* captures developer productivity in two ways. First, new lines committed to a project are recorded in the metric. Second, a line modified is recorded as a *lines_added* and *lines_removed*. While this metric captures development after a file has been created, it misses a critical fact: a file has a size when it is committed that is not recorded in *lines_added*. This size represents development effort that was performed outside of the purview of CVS and is recorded in *initial_size*. These initial sizes represent the vast majority of the size of the project.

As a result of the omission, the analysis in [26] was calculated upon a small, biased subset of the available data—only revisions that modified existing files were included. It is likely that the analysis is biased towards later stages of development and maintenance when changes are more likely to modify existing files than they are to commit new files. Bug fixes and modifications can inflate language entropy because they require few changes to numerous

files. A web developer who adds a field of information to an ecommerce site may make single line changes in SQL, Java, CSS, and HTML. In initial development, on the other hand, a single developer may work on the Java file for an extensive period of time, to the exclusion of other languages. If the initial development is committed as a whole, and not as incremental changes, it would not have been included in the language entropy analysis.

2.5 Insights and Conclusions

While we've tried to articulate a series of caveats with respect to the use of SourceForge data for evolutionary analysis, we don't intend to send an overly negative message. With proper awareness and appropriate methodological adjustments, SourceForge data is a fertile source from which to draw information and conclusions about open source development. However, these conclusions must be tempered by taxonomic caveats based on a full understanding of the problems we've identified in this paper.

2.5.1 Mitigation of Project Problems

Unnatural project growth occurs in a high percentage of projects, depending on the definition of "unnatural." Several questions must be answered in order to form a workable understanding:

1. What is the threshold of *Daily Commit* beyond which we can comfortably conclude that the data is not fully representative of the development effort?
2. How should that threshold of *Daily Commit* change based upon the number of authors contributing at a particular point in the project life cycle?
3. Is there a model that will expose, with high probability, projects for which the data is of sufficiently fine granularity that researchers can draw conclusions about developmental and collaborative practices without requiring heavy qualification of the results due to data sparseness?

Answering these questions will provide researchers far more confidence in their results than is currently advisable.

2.5.2 Mitigation of Author Problems

Author problems may be slightly easier to overcome than project problems. Unlike project data, author data is derived from a single source, although there is some question whether or not source code always represents a single developer. Temporal analysis of a developer's activities readily reveals unusual spikes in development, such as those at the beginning of Figure 2.7. After the initial spike, it appears that 'keess' has a somewhat normal commit pattern which could be used for analysis. However, further work is required to ensure that this assertion is correct.

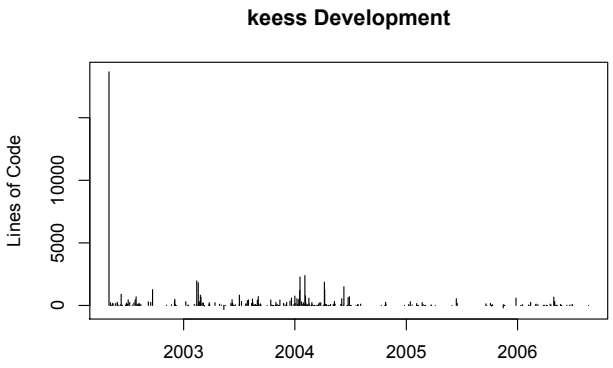


Figure 2.7: Development behavior of 'keess.'

2.5.3 Analytic Adaptations

SourceForge provides a wealth of data that, like other data sources, can easily be misinterpreted due to biases, masking, ambiguity, and sparseness. As researchers, by identifying pitfalls in the data and methods of compensation we increase the applicability of our results. These methods of analysis fortify our results against data irregularities and validate our exploration in this realm of open source software development.

2.5.4 Impact on the Original Study

The original study likely suffers from inflated language entropy numbers due to the cliff walls discussed herein as well as the exclusion of the *initial_size* values. As discussed in Section 2.4, these omissions probably bias the study towards later stages of development when incremental changes are more prevalent than new source files. Ultimately, the study needs to be reevaluated in light of the findings discussed in this work.

2.5.5 Differentiated Replication

This study highlights the benefits of differentiated replication. The authors of the original study were satisfied that their work accurately summarized author programming language usage in SourceForge. However, when analyzed from a different angle—project development rather than single developer activity—it is apparent that they failed to account for several anomalies in the data. Although the authors in the original study strove to provide an unbiased, complete analysis of the data, the domain is simply too large to understand through a single study. Replication affords researchers new avenues and veins of exploration in partially explored areas and is a valuable tool to broaden and deepen understanding in a domain.

Chapter 3

Trends That Affect Temporal Analysis Using SourceForge Data

SourceForge is a valuable source of software artifact data for researchers who study project evolution and developer behavior. However, the data exhibit patterns that may bias temporal analyses. Most notable are *cliff walls* in project source code repository timelines, which indicate large commits that are out of character for the given project. These cliff walls often hide significant periods of development and developer collaboration—a threat to studies that rely on SourceForge repository data. We demonstrate how to identify these cliff walls, discuss reasons for their appearance, and propose preliminary measures for mitigating their effects in evolution-oriented studies.

3.1 Introduction

As organizations construct software, they naturally and inevitably generate artifacts, including source code, defect reports, and email discussions. Artifact-based software engineering researchers are akin to archaeologists, sifting through the remnants of a project looking for software pottery shards or searching for ancient software development burial grounds. In the artifacts, researchers find a wealth of information about the software product itself, the organization that built the product, and the process that was followed in order to construct it. Further, researchers gain the ability to view artifacts not only as static snapshots, but also from an evolutionary perspective, as a function of time. [11, 32]

Artifact-based research methods help resolve some of the limitations of traditional research methodologies. For instance, data collection is often the most time consuming

research activity. Leveraging data that is already resident in repositories—collected as a byproduct of production processes—can save a significant amount of time and effort. Using artifact data, researchers can address software evolution questions in a matter of months that would otherwise require longitudinal studies to be conducted over multiple years. Further, since artifact data is a product of "natural" development processes, research procedures are less likely to have tainted it. Generally speaking, the act of observing human-driven processes can cause those processes to change. Since observational studies are designed to analyze a process "in the wild," any tampering with the context of that process threatens the primary assumption of the study. Therefore, artifact-based research significantly reduces the likelihood that a study's procedure will impact the observed processes.

Despite its benefits, artifact-based research suffers from limitations. For instance, artifact data is temporally separated from the processes that produced it. Therefore, researchers must reconstruct the context in which the artifacts were originally created. Additionally, since artifact data is removed from its original context, identifying the development attributes actually recorded in the data can be difficult. It is challenging enough to ensure that measurements taken for a specific purpose actually measure what they claim to measure [9]. It is all the more difficult (and necessary), therefore, to validate artifact data, which is generally collected without a targeted purpose.

Understanding the limitations of artifact data is integral to the agendas of several research communities (e.g., FLOSS, MSR, ICSE, and WoPDaSD) and is an important step toward validating the results of numerous studies (e.g., [6, 12, 18, 25, 31, 36, 40]). In this paper we examine some of the limitations of artifact data by specifically addressing the applicability of SourceForge data to the study of project evolution.

We select SourceForge data for several reasons. First, although thousands of software projects produce millions of artifacts each year, many of those projects are conducted behind closed doors, where access to data is prohibited by corporate and/or government policies. Consequently, projects for which the artifacts are freely available are generally produced

under the banner of Open Source Software (OSS). Although some argue that the OSS model is fundamentally different from industrial software development models [35], recent studies suggest that the two may not be as different as originally thought [7, 15]. Further, as one of the largest OSS hubs, SourceForge hosts thousands of projects—providing extensive data on thousands of mature projects [13]. These projects are also stored in a consistent format (formerly CVS for source code, but more recently SVN), which allows researchers to compare measurements across projects and to reuse mining techniques across studies. SourceForge data is important to the work of a large and growing community of several hundred researchers.[1]

Our concerns regarding the limitations of SourceForge data originated from efforts to replicate the results of a previous study [25, 26]. This effort led us to analyze the growth patterns of SourceForge projects. As we visualized the evolutionary development of SourceForge projects, we discovered that temporal studies within SourceForge are not as straightforward as they at first appear, and that measuring project evolution in SourceForge is fraught with complications. Mitigating the limitations we discuss in this paper is essential to validating the results of studies that examine the evolutionary aspects of SourceForge data.

Objective: *Understand the limitations of using SourceForge data to address software evolution research questions.*

3.2 Problems

SourceForge data presents several problems that can bias or invalidate evolutionary analyses. In this section, we address three of these problems: Non-Source Files, Cliff Walls, and High Initial Commit Percentage. These problems particularly affect calculations that utilize project growth measures based on lines of code added or removed. For our analysis we examine 9,997 Production/Stable or Maintenance phase projects stored in CVS on SourceForge and extracted in October of 2006 [12].

[1]The number of subscribers to the SRDA (SourceForge Research Data Archive) currently exceeds 100 [38]. The actual number of researchers engaging SourceForge data is likely several times that.

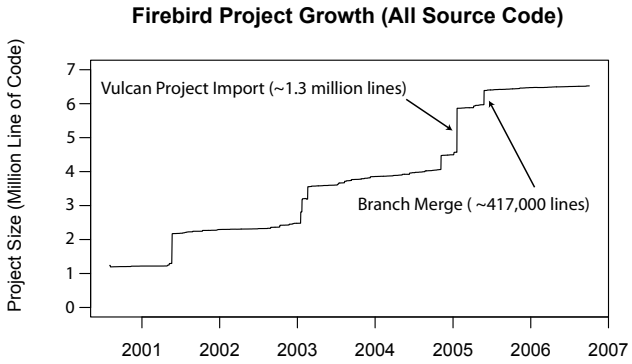


Figure 3.1: Growth of Firebird over time.

3.2.1 Non-Source Files

Many of the text-based files in projects on SourceForge are not source code files. Examples include documentation files, XML-based storage formats, and text-based data files such as maps for games. It is unclear how to compare source code production with production of non-source text-based files. In order to accurately analyze author and team contributions to projects, we filter out these non-source files.

Most file extensions occur infrequently in SourceForge data. Of the 21,125 unique file extensions identified, 195 were classified as common source code extensions. Studies of source code development should limit themselves to these source code files. Our treatment of additional data problems herein presumes a set of projects filtered under these criteria.

3.2.2 Cliff Walls

Many projects in our data set exhibit stepwise growth patterns which we refer to as "Cliff Walls." These monolithic commits appear as vertical (or near vertical) lines in an otherwise smooth project growth timeline (see Figure 3.1). In our analysis we group commits into days to identify cliff walls programmatically.

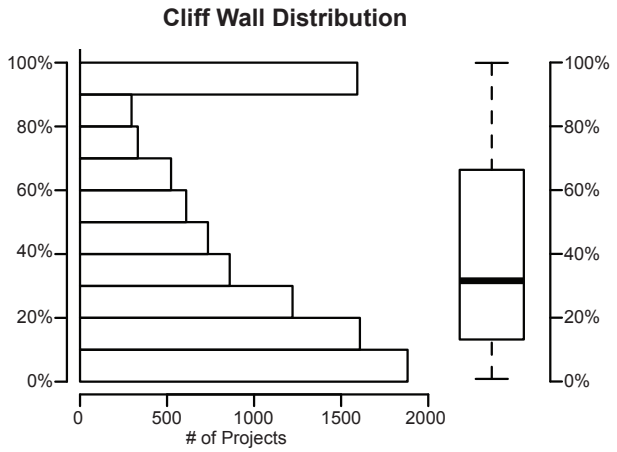


Figure 3.2: Distribution of projects by largest cliff walls. One outlier has been removed.[2]

Anomaly Description

The average size of the largest cliff wall for a project is 41.8% of the total size of the project. The median is 30.8%, meaning that half of the projects in our data set have a cliff wall that is nearly a third of the project size. Figure 3.2 shows the distribution of projects by largest cliff walls as a percentage of total project size as of the date of data collection. The histogram represents the number of projects discretized by their largest cliff wall. For example, in the $0-10\%$ bin there are 1,882 projects, meaning that for these 1,882 projects the largest cliff wall is $0-10\%$ of the project size.

Cliff walls appear in all phases of project growth. In Figure 3.1 we see monolithic commits throughout the studied life cycle of the project. However, in the Java eXPerience FrameWork project (JXPFW), we only see this pattern at the beginning (see Figure 3.3). After the initial source commit (2 1/2 years after the project was created) JXPFW appears to grow normally.

[2] We removed one outlier from the data set when creating these images. The "Codice Fiscale" project had a large commit of 14,158 lines of code of which 13,686 were removed the following day. The total size of the project was only 4,530 on the date our data was gathered. As a result, the project has a cliff wall percentage of 312.54%. All other projects in our data set lie between 0% and 100%.

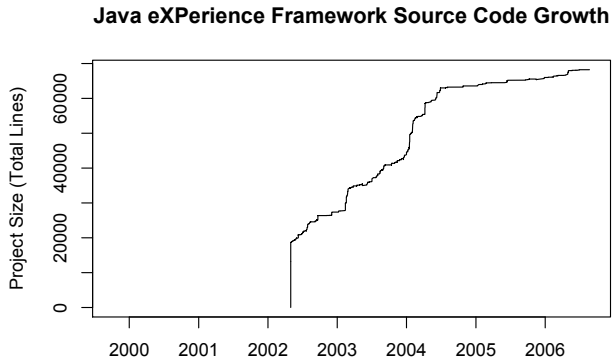


Figure 3.3: Growth of the Java eXPerience FrameWork over time.

Problems in Analysis

Cliff walls can cause severe biases in analysis of project evolution. If a large commit comprises several months of software development activity, productivity metrics will be erroneously high for the time period prior to the commit. In addition, developers will wrongfully appear to be inactive for the previous time periods.

A cliff wall may appear in the data for a number of reasons. In Section 3.3 we discuss four of those reasons.

3.2.3 High Initial Commit Percentage

Most of the projects in our data set grow almost exclusively by initial commit size (the size of files when they are initially checked into CVS). The size associated with this commit, in lines of code, is distinct from lines of code committed to (or deleted from) a preexisting file.

Anomaly Description

Initial Commit Percentage (ICP) is the percentage of the total size of the project that is made up of initial commits. Figure 3.4 shows that most projects have a high ICP. In fact, 83.6% of projects have an ICP of 80% or higher. This would seem to make sense given the power law distribution of projects sizes and the assumption that a big commit to a smaller project has a more pronounced effect (see Figure 3.5; note the log scale on the y-axis). However, this

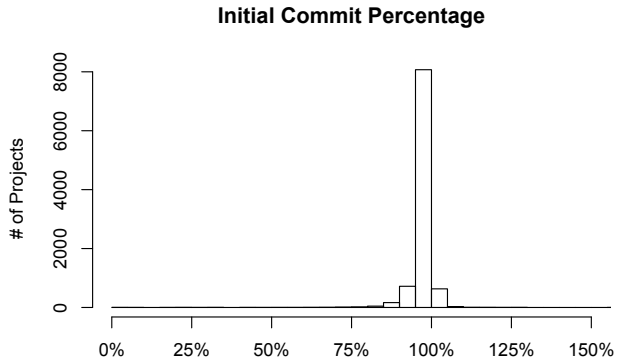


Figure 3.4: Distribution of projects by Initial Commit Percentage.

distribution holds, with small variation, regardless of project size (see Figure 3.6). High ICP indicates that revisionary changes to existing files constitute a small percentage of project growth.

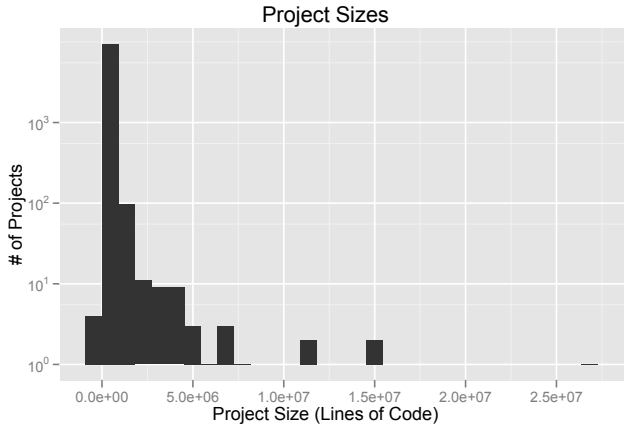
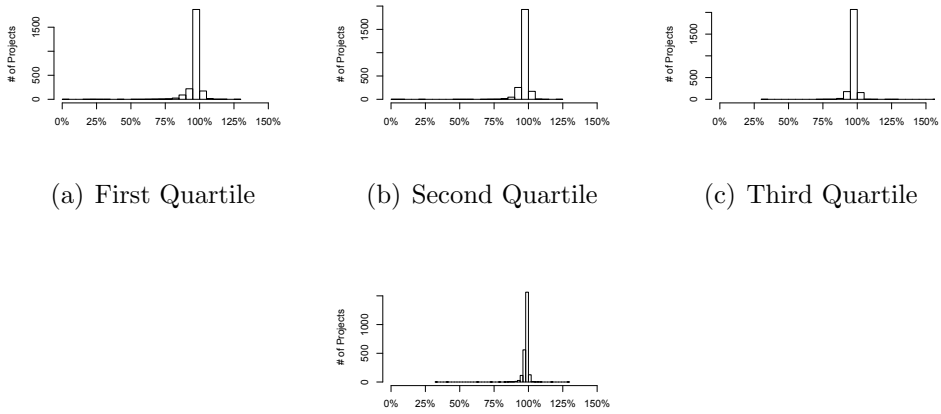


Figure 3.5: Project sizes.

Problems in Analysis

High ICP does not, by itself, threaten appropriate and effective analysis. However, many of the causes of high ICP may introduce threats to validity, as discussed in Section 3.3.



(a) First Quartile (b) Second Quartile (c) Third Quartile

(d) Fourth Quartile

Figure 3.6: Distribution of project by Initial Commit Percentage discretized by project size quartile.

3.3 Reasons for Problems

Although there are many possible causes for the anomalies mentioned in Section 3.2, we identify four that we believe to be chief among them: Off-line Development, Auto-Generated Files, Project Imports, and Branching. Our inclusion of these four should not be construed as dismissive of other causes. Instead, these four causes represent, in the opinion of the authors, the largest contributors to the aforementioned anomalies in projects on SourceForge *as a whole*. Other factors may be more important than these when examining an individual project.

3.3.1 Off-line (Internal) Development

Many projects in our data set are committed as finished, monolithic entities. After the initial commit the authors commit infrequently and in large chunks. They do not commit frequent, incremental changes that capture development at a fine granularity. In essence, these projects use SourceForge as a delivery mechanism rather than a collaborative development environment. We postulate that a few key factors may explain this phenomenon.

The first factor is that it may be easier or preferable for co-located developers to collaborate via local tools, such as a locally hosted repository, or tools that are unavailable on SourceForge, such as GIT. These teams of "volunteer"[3] developers are free to use a separate "Repository of Use" and utilize SourceForge as a "Repository of Record" [22].

Second, projects with large corporate sponsors may be primarily developed in-house within a local development framework. When an established development organization begins or adopts an open source project it is logical to assume that the organization will continue to operate as it has in the past. This assumption precludes integrating SourceForge into the collaboration and build process. Instead, SourceForge becomes a release mechanism, rather than an integral part of the development process.

Lastly, some projects use gatekeepers as a means of quality control. These first tier authors are responsible for reviewing source code before it can be committed to the repository. In benign cases the second tier author creates a branch (discussed in Section 3.3.4) within the SourceForge CVS repository which the gatekeeper inspects before merging it into the trunk. The branch preserves all of the temporal data relating to the development efforts of the second tier author. However, in other cases this review process occurs outside the purview of the repository. In essence, there exist only first tier authors who commit all of the changes to the repository, regardless of who actually produced them.

Each of these occurrences produces commits that are bursty and lossy. Both outcomes result from aggregating an extended work period into a single recorded event. Instead of recording events throughout the work period, and thereby retaining finer grained development information, authors commit at the end of a protracted development effort. Consequently, cliff walls are evident in the data and the ICP is high.

[3]We use the term "volunteer" in deference to other researchers who categorized open source developers as such. However, many key "volunteers" are on the payroll of open source projects, which calls into question the use of the term "volunteer".

3.3.2 Auto-Generated Files

While the bulk of code in source code repositories is written manually, developers can use several tools to automatically generate copious amounts of source code (e.g., GUI design tools, lexical analyzers, and program translators [30]). The presence of auto-generated code is a source of uncertainty when analyzing data extracted from SourceForge. Tools that generate such code often produce large quantities of code very quickly, which is attributed to whomever commits it. The result is that factors such as project size, productivity, cost, effort, and defect density are often inaccurate [30]. We believe that commits containing auto-generated code contribute to the presence of the cliff walls we have identified.

Unfortunately, the problems created by auto-generated code in SourceForge are not easily resolved. Due to the variety of tools generating such code, the existence of a one-size-fits-all solution for identifying auto-generated code is unlikely. Uchida et al. suggest that code clones may be useful in the detection of auto-generated code. Their study found that auto-generated code was a common cause of code clones in a sample of 125 packages of open source code written in C [37]. Further investigation is needed to substantiate the utility of code clones as an indicator for auto-generated code. However, given the computational intensity of current methods of identifying code clones, their detection is unlikely to be a panacea.

3.3.3 Project Imports

In Figure 3.1 we see a cliff wall labelled "Vulcan Project Import." This cliff wall represents an import of slightly over 1.3 million lines of code from a project named *Vulcan* into *Firebird*. Imports represent development that occurred outside of the current repository. Depending on their size, they can result in cliff walls and high ICP. All code committed through an import is considered an initial revision, rather than a revisionary change.

3.3.4 Branching

The CVS version control system supports branching, a feature that enables concurrent development of parallel versions of a project. However, Zimmermann et al. note that branch merges in CVS cause undesirable side-effects for two main reasons: they group unrelated changes into one transaction and they duplicate changes made in the branches [42].

One such side effect materializes when researchers attempt to estimate project size through analysis of CVS logs. Changes made in a branch are counted twice: first when they are introduced into the branch, and second when the branch is merged, resulting in a project size estimate inflated by as much as a factor of two. A portion of cliff walls can also be explained by merges. A merge combines all transactions on a branch that have not previously been merged into one transaction. If a significant amount of development has taken place prior to the merge, the merge will likely appear as a large cliff wall. In Figure 3.1 the cliff wall labeled "Branch Merge" is a merge, not new code.

Merges can also falsely inflate measures of author contributions. All of the changes reflected in the merge transaction are attributed to the developer who performs the merge, regardless of whether or not that author actually produced any of those changes. If researchers do not take measures to correctly handle merges, analysis results may be unreliable.

3.4 Solutions

In order to derive useful, accurate results in temporal analysis of projects hosted on SourceForge we must identify methods of mitigating the problems and associated causes that we've identified. Fortunately, for most of these issues, complete or partial solutions are available and computationally solvable. However, for some of these issues, a scalable solution is not readily apparent.

3.4.1 Identify Merges

In Section 3.3.4 we discuss some of the difficulties that merges create for those studying SourceForge data. However, certain approaches may allow researchers to overcome issues caused by merges.

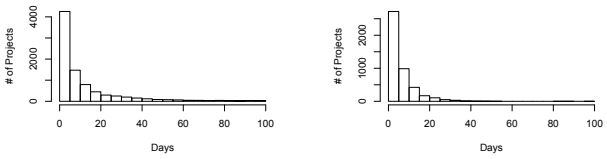
Zimmermann et al, suggest a very simple approach to identifying merge transactions wherein researchers manually examine each transaction for which the log message contains the word "merge" and determine if the transaction is indeed the merge of a branch [42]. There are drawbacks to this approach. First, it is unknown what percentage of merges actually include the word "merge" in the log message. It is possible that researchers may overlook a significant number of valid merges due to custom log messages that use synonyms for "merge" or that remove the word altogether. Additionally, manual approaches scale poorly as the size of the data set increases. As a result, this method may be excessively time consuming for large quantities of data.

Fischer, Pinzger, and Gall suggest a different approach for identifying merges in CVS. The authors utilize revision numbers, dates, and diffs between different revisions of a source file [16]. This approach is computationally intensive and may not scale to studies of large sets of projects.

We suggest the possibility of a third method, that of simply assuming that all revisions containing the "merge" keyword are merges. This is the fastest method that we have yet identified, but would also likely suffer in terms of accuracy. Future work is required to establish the best method(s) for identifying merges in CVS, in terms of speed and accuracy.

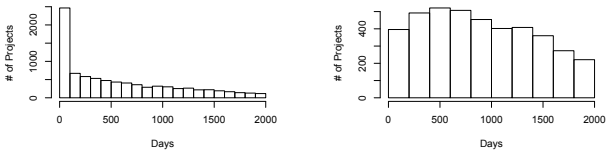
3.4.2 Author Behavior

One way to identify project records that contain fine grained evolutionary data is to filter for projects that have authors who "commit early, commit often." *Frequency of commits* is a metric that captures this behavior. Figure 3.7 illustrates the distribution of projects by commit frequency. We also show the distribution for projects with more than 40 commits



(a) All Projects (b) ¿ 40 commits

Figure 3.7: Distribution of projects by frequency of author commits.



(a) All Projects (b) ¿ 40 commits

Figure 3.8: Distribution of projects by project life span: the time between the first and the last commit in a project.

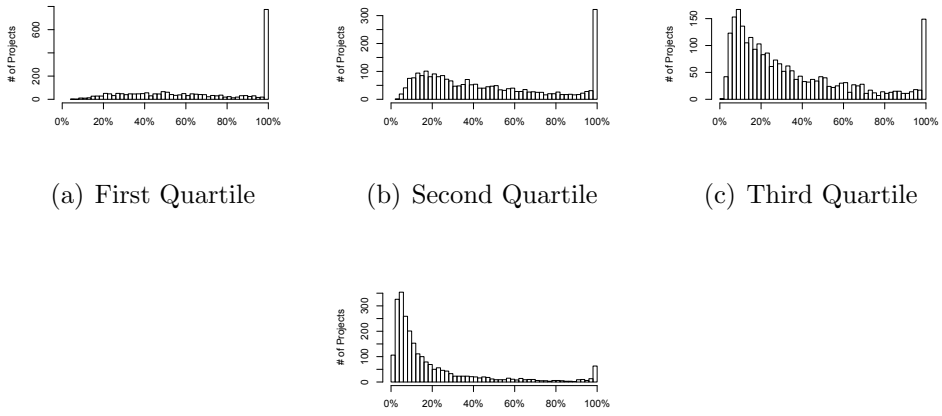
to show that the graphic is not overly biased by small projects that are completed quickly. There appear to be plenty of projects that satisfy a high *frequency of commits* requirement. In Figure 3.8 we see that by limiting the data set to projects with more than 40 commits we also get rid of most of the short-lived projects.

3.4.3 Project Size

Small projects have a much higher occurrence of large cliff walls than large projects. Figure 3.9(a) illustrates that in the first quartile of project sizes (0 to 12,307 lines of code) 31.8% of projects are almost entirely made up of one monolithic commit. Interestingly, all of the histograms in Figure 3.9 have a spike at 100%. However, 3.9(b), 3.9(c), and 3.9(d) have successively greater area under the curve towards 0% (see Table 3.1 for quartiles). This

0%	25%	50%	75%	100%
0.0	12,307.5	58,517.0	271,848.2	117,147,667.0

Table 3.1: Project Size Quartiles (Lines of Code)



(a) First Quartile (b) Second Quartile (c) Third Quartile

(d) Fourth Quartile

Figure 3.9: Distribution of projects by largest cliff wall as a percentage of project size. See Section 3.2.2 for a discussion of how to read these histograms.

suggests that in the second, third, and fourth quartiles there are many projects that have small, incremental commits and may be appropriate for temporal analysis.

3.5 Insights

Artifact-based evolutionary research of projects on SourceForge can yield unbiased results corroborated by thousands of projects. However, we must choose projects cautiously to avoid the pitfalls identified in this paper. Further work is necessary to develop a taxonomy of projects in this ecosystem to better understand how to choose projects automatically.

Additionally, analysis of the interaction between available meta variables may help expose projects that capture a fine-grained development effort. Figures 3.7 and 3.8 suggest that a significant subset of medium to large projects on SourceForge can be used for evolutionary analysis. We hope that as we further refine our methods of selecting projects we can develop an automated procedure for choosing projects that have the finest possible detail in their revision history.

Chapter 4

Cliff Walls: An Analysis of Monolithic Commits Using Latent Dirichlet Allocation

Artifact-based research provides a mechanism whereby researchers may study the creation of software yet avoid many of the difficulties of direct observation and experimentation. However, there are still many challenges that can affect the quality of artifact-based studies, especially those studies examining software evolution. Large commits, which we refer to as "Cliff Walls," are one significant threat to studies of software evolution because they do not appear to represent incremental development. We used Latent Dirichlet Allocation to extract topics from over 2 million commit log messages, taken from 10,000 SourceForge projects. The topics generated through this method were then analyzed to determine the causes of over 9,000 of the largest commits. We found that branch merges, code imports, and auto-generated documentation were significant causes of large commits. We also found that corrective maintenance tasks, such as bug fixes, did not play a significant role in the creation of large commits.

4.1 Introduction

Artifact-based software engineering research may in some respects be compared to archaeology, a field that has been defined as "the study of the human past, through the material traces of it that have survived"[5]. Much like archaeologists, empirical software engineering researchers often seek to understand people. The software engineering researcher, while not isolated from a target population by eons, faces other obstacles that often make direct observation

impossible. Many organizations are loath to allow researchers through their gates, in an effort to protect trade secrets or merely to hide shortcomings. Even in cases where researchers are allowed to directly observe engineers building software, the Hawthorne effect threatens the validity of such observations. Time investment and organizational complexity are also issues that pose problems in software engineering research. Direct observation requires significant time investment, making it impossible for a single researcher to observe everything that takes place within a given software organization.

As a result of these barriers, software researchers, like their archaeologist counterparts, take advantage of artifacts—work products left behind in software project burial grounds. Artifacts are collected after the fact, minimizing the confounding influence of the presence of a researcher. Artifacts also help researchers deal with the requirements of studying complex organizations. By leveraging artifacts of the software process, researchers are able to study thousands of pieces of software in a relatively short period of time, an otherwise impossible task.

The open source movement is particularly important to software engineering research, since project artifacts, such as source code, revision control histories, and message boards, are openly available to software archaeologists. This makes open source software an ideal target for researchers with a desire to understand how software is built.

4.1.1 Threats to Artifact-based Research

Unfortunately, the study of artifacts in software engineering is not all sunshine and double rainbows; serious challenges threaten the results of artifact-based research involving open source software projects (such as those hosted on SourceForge). Since artifact data is examined separate from its original development context, identifying the development artifacts actually recorded in the data can be difficult. It is challenging enough to ensure that measurements taken for a specific purpose actually measure what they claim to measure [9]. It is all the

more difficult (and necessary), therefore, to validate artifact data, which is generally collected without a targeted purpose.

Many phenomena can easily be overlooked by software archaeologists, such as auto-generated code, the presence of non-source code files in version control, and the side effects of branching and merging. Understanding the limitations of artifact data represents an important step toward validating the results of numerous studies (for example, [6, 12, 18, 25, 31, 36, 40]). Despite on-going efforts to identify and mitigate the limitations of artifact-based research, new threats are constantly emerging.

The focus of this paper is one such threat—the presence of monolithic commits in open source repositories. A close look at the version control history of many projects in SourceForge reveals some worrisome anomalies. Massive commits, which we refer to as "Cliff Walls," appear with alarming frequency. One investigation of Cliff Walls in SourceForge found that out of almost 10,000 projects studied, half contained a single commit that represented 30% or more of the entire project size [28]. These Cliff Walls indicate periods of unexplained acceleration in development, threatening some common assumptions of artifact-based research, especially for studies of project evolution. Cliff Walls thwart attempts to tie authors to contributions, mask true development history, and taint studies of project evolution. We must better understand Cliff Walls if we are to paint an accurate picture of software evolution through the use of artifacts.

4.2 Cliff Walls

In [28] we introduced the concept of "Cliff Walls" as unexpected increases in version control activity over short periods of time (see Fig. 5.1). In the most extreme cases, Cliff Walls represent millions of lines of source code contributed to the repository in less than a day. Such activity is problematic for researchers, especially those investigating the evolution of software, because such sudden surges of source code commits cannot possibly be the result of common incremental development activities. Even a team of "supercoders" would be

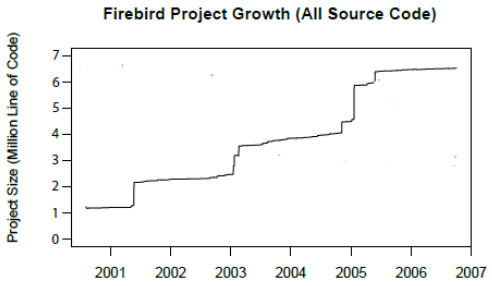


Figure 4.1: Cliff Walls in the Firebird Project

hard-pressed to produce such massive quantities of code in a short period of time, all the more impossible for the single "author" to which a version control commit is attributed. We are forced to ask what these "Cliff Walls" truly represent.

4.2.1 Definitions

When examining contributions to version control systems, it is logical to discuss them in the context of "commits." However, some researchers have defined "commit size" in terms of the number of files changed in a commit [19–21], while others treat the size of a commit as a function of the number of LOC added, removed or modified [4].

Within the context of this study, a *commit* refers to the set of files and/or changes to files submitted simultaneously to a version control system by an individual developer. Since we are primarily concerned with the growth of projects over time, *Commit size* is defined as the total number of lines added or removed from all source code files in a given commit, which allows us to perceive code growth (or shrinkage) within a project. For example, if an existing line in one file is modified, and no other changes are made, the size of the resulting commit would be zero LOC. In this study, we include code comments in the measurement of commit size, allowing us to detect code growth attributable to the insertion of comments within source files.

We use the term *Cliff Wall* to describe any single commit with size greater than 10,000 lines of code (LOC). We believe this threshold to be sufficiently high that it avoids capturing

most incremental development contributions for individual authors. While it is possible that an extremely productive developer could produce 10,000 lines of code in a period of days or weeks, it is unlikely. Additionally, the methods employed in this study should allow us to identify instances where the threshold fails.

4.2.2 Commit Taxonomies

The ability to distinguish between different types of cliff walls is critical for many artifact-based studies. For example, a researcher attempting to measure developer productivity on a project would likely want to be able to distinguish between large commits caused by auto-generated code and branch merges, as they must be handled differently when calculating code contributions.

In [21], the authors present a taxonomy used to manually categorize large commits on a number of open source software projects. Hattori and Lanza also present a method for the automatic classification of commits of all sizes in open source software projects [19]. The taxonomies developed in these studies focus on classifying the type of change made (for example, development vs. maintenance). Both these studies concluded that corrective maintenance activities, such as bug fixes, are rarely the topic of large commits. These studies also found that a significant portion of large commits were dedicated to "implementation activities." The authors consider all source code files added to the repository to be the result of implementation activities.

In [4], the authors categorize commits from open source projects into three groups, using a rough heuristic based on commit size. Their study makes no attempt to analyze commit log messages. Rather, they divide each commit into one of three groups, based on the commit size: 1) single individual developer contributions, 2) aggregate developer contributions, and 3) component or repository refactoring or consolidations.

In our study, we seek to identify specific behaviors that play a significant role in the creation of monolithic commits. In the next section, we discuss the methods that we will apply in our search for the causes of Cliff Walls.

4.3 Latent Dirichlet Allocation

Latent Dirichlet Allocation (LDA) is an unsupervised, hierarchical Bayesian topic model for analyzing natural language document content that clusters the terms in a corpus into topics [8]. In LDA, the model represents the assumption that documents are a mixture of these topics and that each topic is a probability distribution over words. In short, LDA allows us, without any knowledge engineering, to discover the topical content of a large collection of documents in an unsupervised fashion. Given a corpus of documents such as our commit log messages, inference techniques such as Gibbs sampling or Variational Bayes can be used to find a set of topics that occur in that corpus, along with a topic assignment to each word token in the corpus.

A common technique for visualizing the topics found using LDA is to list the top n most probable terms in each topic. For example, one of the topics we found using LDA that concerns bug correction, consisted of the following terms:

fix bug fixes crash sf problems small quick closes blah deleting weapon decoder lost hang weapons delphi noted led

When brevity is required, it is also common to refer to a topic by just the first two or three most probable terms, sometimes in hyphenated form. Using this notation, the above topic could be referred to as the "fix-bug" topic. Both of these methods for indicating a particular topic (word list and first words) are used throughout this paper.

Because LDA also assigns a topic to each individual token within a commit message, a single commit can, and usually does, contain many topics. This is one benefit of using LDA: each "document" may belong to multiple classes. Such is not the case with many of

the supervised classification methods, which typically assign each document to a single class. In the case of commit log messages, allowing for multiple topics in a single message allows us to conduct a more fined-grained analysis.

The approach taken by LDA is an attractive alternative to other classification and clustering methods. Supervised classification methods require a training set of "tagged" data, for which the appropriate class for each instance has previously been specified. In many situations this tagged data does not exist, requiring the researcher to manually assign tags to some portion of the data. This is a tedious, time-consuming, and potentially biased process which can be avoided through the use of unsupervised methods such as LDA. Unsupervised methods may also compensate for a measure of "short-sightedness" by the researcher. In supervised methods, since the classes must be predefined by the researcher, it is possible to degrade the usefulness of the model through the omission of important classes, or the inclusion of irrelevant classes. Unsupervised methods avoid some of these errors since no predefined clusters are specified, allowing the data to better "speak for itself." For our analysis, we used the open source tool "MALLET" which includes an implementation of LDA [29].

4.4 Methods

Our data set consists of the version control logs of almost 10,000 projects from SourceForge, acquired in late 2006. This data set has been previously used in a number of studies [12–14, 25–28]. The logs for all projects in our data set were extracted from the CVS version control system. In creating the data set, the original authors filtered projects based on development stage; only projects labeled as Production/Stable or Maintenance were included in the data set. For further description of the data, see [12]. In calculating commit size, we excluded non-source code files, based on file extension. Over 30 "languages" were represented in the final data set, including Java, C, C++, PHP, HTML, SQL, Perl and Python.

Because CVS logs do not maintain any concept of an atomic multi-file "commit" it was necessary to infer individual commits. We utilized the "Sliding Time Window" method

introduced by Zimmerman and Weißgerber [41]. This resulted in a set of almost 2.5 million individual commits, extracted from over 26 million file revisions. Applying our pre-defined threshold of 10,000 LOC yielded over 10,000 Cliff Walls. We also found that a number of the commits contained log messages that were uninformative. Commits with empty log messages or with **"empty log message"** were removed from the data to prevent degradation in the quality of topics identified. The resulting set contained 2,333,675 commits, with 9,615 Cliff Walls. We later removed other uninformative commits (see discussion in Sec. 4.6), ultimately resulting in the exclusion of 6.6% of commits in our data set due to log messages that conveyed no information about the development activities they represent. A disproportionate number of the commits removed were Cliff Walls (an ultimate exclusion of 14.8% of all Cliff Walls). Additionally, very common English adverbs, conjunctions, pronouns and prepositions belonging to our "stop-word" list were removed from the commit messages in order to ensure the identification of meaningful topics.

The LDA algorithm, as implemented in MALLET, requires three input parameters: the number of topics to produce in its analysis, the number of iterations, and the length of the burn-in period. In our study, we elected to identify 150 topics with MALLET. The authors Hindle and German identified 28 "types of change" for the commits classified as a part of their taxonomy [21]. Hattori and Lanza, in their study of commit messages, identified 64 "keywords" that were used to classify commits [19]. These prior results gave us reason to believe that 150 topics would be a sufficient number to capture the motivations behind the commits in our data set, with an appropriate level of detail.[1]

As one of the steps to understanding the Cliff Wall phenomenon, we compare the most prevalent topics found in Cliff Wall commits to those found in the entire body of commits. Instead of running LDA separately on the two subsets of our data, we run LDA once on all of

[1]The other two parameters, number of iterations and length of burn-in period, are required by the Gibbs Sampling inference method employed by MALLET. We refer the reader to [39] for a description of LDA as it is implemented within MALLET, including a description of Gibbs Sampling. For these two parameters we used the default values provided by MALLET; 1,000 iterations with 200 dedicated to burn-in. Further work should investigate the possibility of more appropriate values for all three parameters, as discussed in Sec. 4.6.

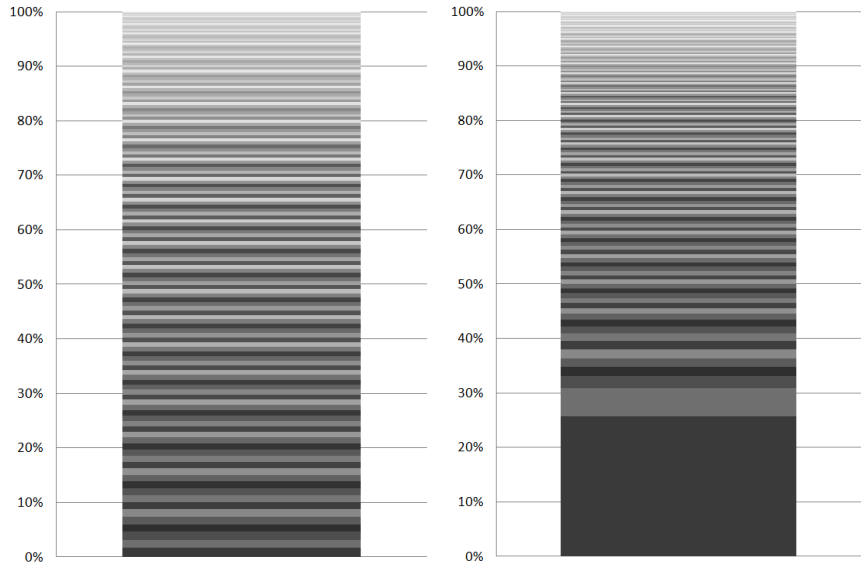


Figure 4.2: Topic distribution for All Commits (left) and Cliff Wall Commits (right)

the data and then filter the results to gain a picture of Cliff Walls in contrast to All Commits. This approach ensures that the topics found are consistent across both groups, which helps yield a meaningful comparison.

4.5 Analysis & Discussion

Figure 4.2 provides a first glance at some of the variation exhibited by Cliff Walls. In these graphs, each horizontal bar represents one of the 150 topics generated. The thickness of each bar represents the proportion of tokens in the entire corpus of commits that were assigned to that topic. Commit log messages are fairly evenly distributed over the topics for the general population of commits. However, a small number of topics are considerably more prevalent in the large commits. Tables 4.1 and 4.2 list the 15 most prevalent topics for all commits and Cliff Wall commits.

Similarly, the tag clouds in Figs. 4.3, 4.4, and 4.5 begin to give us an idea of the most common topics for our two groups of interest.[2] Each tag in the cloud represents a topic that has been summarized using the "first words" method described in Sec. 4.3. Like a stratum

[2]All tag cloud images were generated using Wordle (www.wordle.net).

Figure 4.3: Topic Tag Cloud: All Commits

from Fig. 4.2, each tag is sized based on the proportion of tokens belonging to that topic. Thus, the largest tag in Fig. 4.4, "initial-import," is also the largest stratum in the bar chart for Cliff Walls. Tag position and color do not convey any additional information. Figure 4.5 is an alternate view of Fig. 4.4, with the dominant "initial-import" topic removed to improve readability. These images provide an overall view of the topics and their proportions for the two groups of interest. We next discuss some of the most prevalent topics and their interpretations.

One of the goals of this paper is to compare the topics produced by LDA with previously hypothesized causes of Cliff Walls. We examine each of these causes to see if LDA is able to identify them as prominent features of large commits. We also examine some of the causes for Cliff Walls that were previously overlooked, but were consequently suggested by LDA. In pursuit of these goals, we have found two views of the data that provide insight into the causes of Cliff Walls: Overall Topic Proportion and Topic Relative Rank. We discuss these in the next two subsections.

4.5.1 Overall Topic Proportion

Tables 4.1 and 4.2 display the 15 most prevalent topics in the Cliff Wall and All Commits groups, as determined by proportion of tokens belonging to that topic. For example, the topic "version-release" in Tbl. 4.1 has a proportion of 1.75%, suggesting that 1.75% of all the

Figure 4.4: Topic Tag Cloud: Cliff Walls

Figure 4.5: Topic Tag Cloud: Cliff Walls ("initial-import" excluded)

words in all of the commit messages in our data set were assigned to this topic by the LDA algorithm. In other words, these tables list the topics most frequently discussed in commit log messages belonging to our two groups of interest. We refer back to these tables frequently throughout this section as we discuss the various causes of Cliff Walls.

4.5.2 Topic Relative Rank

Additional insight into Cliff Walls can be gained through the use of another simple metric. Within each of our two groups, "All Commits" and "Cliff Walls," each topic can be ranked based on its proportion within the group. The difference between a topic's ranking in the two groups is a good indicator of the prevalence of a given topic relative to other topics.

For example, in Tbl. 4.1 we see that "initial-import" is the 5th ranked topic. In Table 4.2, the same topic is ranked 1st, for a rank difference of +4. In contrast, as we see from Tbl. 4.3, the topic "cvs-sync" holds ranks of 101 and 4, resulting in a rank difference of +97. In essence this means that, relative to other topics, "cvs-sync" is discussed more frequently within Cliff Wall commits than it is for the general population of commits.

It is important to note that the difference between topic ranks is not synonymous with a similar difference in proportion. The difference between proportions for the "initial-import" topic is an approximately 25% increase for the Cliff Walls group. This is a very large change in proportion which results in a relatively small difference in rank. It is even possible, given the distinct distributions of our two groups (see Fig. 4.2) that a *negative* change in proportion could still result in a *positive* rank difference.

4.5.3 Code Imports

A "Code Import" occurs when a significant amount of source code is added to the repository. Code Imports differ from "Off-line Development" in that the code added was not developed as part of the project of interest, but instead originated from some other project. The

Table 4.1: Top 15 Topics for All Commits

#	Proportion	Key Terms				
1	1.75%	version	release	updated	update	final
2	1.43%	file	branch	initially	added	java
3	1.42%	fixes	minor	small	cleanups	updates
4	1.38%	data	minor	updates	fixes	bugfixes
5	1.38%	initial	import	commit	checkin	revision
6	1.36%	added	comments	comment	documentation	javadoc
7	1.29%	fixed	bug	bugs	incorrect	couple
8	1.28%	added	support	info	extended	basic
9	1.28%	error	message	errors	handling	checking
10	1.26%	removed	code	template	commented	unnecessary
11	1.25%	fix	bug	fixing	small	bugs
12	1.18%	fix	typo	corrected	correct	errors
13	1.17%	fixed	bug	wrong	crash	introduced
14	1.17%	page	link	updated	links	url
15	1.14%	code	cleanup	source	clean	cleaned

Table 4.2: Top 15 Topics for Cliff Walls

#	Proportion	Key Terms				
1	25.74%	initial	import	commit	checkin	revision
2	5.11%	version	release	updated	update	final
3	2.30%	removed	deprecated	sources	constant	imported
4	1.63%	cvs	sync	tabs	real	converted
5	1.60%	error	message	errors	handling	checking
6	1.58%	message	project	messages	error	testing
7	1.51%	merged	merge	head	merging	main
8	1.47%	code	cleanup	source	clean	cleaned
9	1.29%	files	added	dialogs	library	directories
10	1.21%	directory	moved	common	dir	structure
11	1.17%	xml	updated	api	latest	version
12	0.98%	update	added	format	updating	creation
13	0.93%	script	makefile	install	configure	sh
14	0.89%	added	comments	comment	documentation	javadoc
15	0.89%	double	beta	v1	v2	values

Table 4.3: Largest "Positive" Rank Differences

	All Rank	Cliff Wall Rank	Key Terms			
+97	101	4	cvs	sync	update	repository
+97	112	15	beta	v1	v2	v3
+93	131	38	org	cvs	synchronized	packages
+91	98	7	merged	merged	trunk	stable
+91	100	9	files	added	directories	library
+89	99	10	directory	moved	structure	location
+88	142	54	cc	net	users	sourceforge
+80	128	48	module	python	py	libsrc
+73	119	46	system	specific	platform	devel
+71	133	62	http	www	urls	net
+71	105	34	web	site	resource	component
+67	109	42	php	index	http	forum

most common example of a code import is probably the addition of the source code for an externally developed library.

We found a good deal of evidence that Off-line Development is a significant cause of Cliff Walls. As shown in Tbl. 4.2, a few prominent topics (particularly 1, 4 and 9) deal with the first-time addition of files to the repository. The addition of files alone, however, does not indicate a Code Import. Table 4.1 indicates that topic 1 is quite prominent for all commits, because files are constantly being added to version control systems. In the case of Cliff Walls, however, the size of the commit gives us good reason to believe that the files added contain a great deal of code, and therefore do not represent files added as part of incremental development.

Also, a few of the topics do provide additional evidence of Code Imports. Topic 9 contains the term "library" which indicates that this topic relates to the addition of library files to version control. Similarly, the topic "module-python" appears in Tbl. 4.3 as a topic with a much higher relative rank for Cliff Wall commits. Examination of log messages for which this topic had high proportion yielded messages such as "bundle all of jython 2.1 with

Table 4.4: Largest "Negative" Rank Differences

	All Rank	Cliff Wall Rank	Key Terms			
-120	13	133	fixed	bug	wrong	crash
-118	12	130	fix	typo	corrected	correct
-116	7	123	fixed	bug	bugs	incorrect
-105	44	149	button	selection	tab	dialog
-90	23	113	fixed	problems	issue	bad
-89	30	119	string	return	null	true
-80	51	131	variable	global	unused	define
-77	26	103	output	debug	print	messages
-76	49	125	problem	fixed	patch	solution
-75	36	111	size	buffer	limit	bytes
-75	54	129	function	calls	static	inline
-74	3	77	fixes	minor	cleanups	cosmetic

marathon so all python standard library is available" and "Add the Sandia RTSCTS module to the code base." These messages are indicative of Code Imports.

4.5.4 Off-line development

In this paper, we use the term "Off-line Development" to refer to large quantities of code that were developed as part of a project, but for which we have no record. This may be code that was developed without the benefit of version control, or that was developed in a separate repository and then added to the SourceForge CVS repository in a monolithic chunk.

Much of the evidence for Off-line Development is similar to that of Code Imports. Many of the same topics that may refer to code imports ("initial-import," "cvs-sync," and "files-added") could equally be attributed to Off-line Development. Thus it is difficult to distinguish between the true source of many large commits, because it is hard to tell if the files added were developed in conjunction with the current project or separately. Further investigation is required to elucidate the differences between Cliff Walls attributable to Code Import and those due to Off-line Development.

4.5.5 Branching & Merging

Merging is a major factor in the creation of Cliff Walls. The 7th ranked topic for Cliff Walls is a topic dealing with the merging of a branch in the repository. This same topic also appears as one of the largest positive rank differences in Tbl. 4.3. This indicates that not only are merges a significant factor behind the creation of Cliff Walls but also that the merge topic is significantly more prevalent within Cliff Walls than it is for All Commits. In contrast, the "initial-import" topic is one of the highest ranked topics in both groups.

4.5.6 Auto-Generated Code

Topics pertaining to Auto-generated Code are a bit more difficult to identify. The topic "target-generated" appears to capture auto-generated code quite well:

target generated rules rule generate mark reports make generation targets automatically linked policy libraries based generator jam dependencies building

Surprisingly, this topic is of relatively little importance, with rank 67 (Cliff Walls) and 113 (All Commits). Such low ranks would seem to indicate that Auto-generated code does not play a significant role in the explanation of Cliff Walls. We did, however, find another example that suggests that Auto-generated Code may be a more significant factor. We were surprised to find that the topic "added-comments" was the 14th ranked topic for Cliff Walls (see Tbl. 4.2). Non-source code files had been excluded from our study, and code commenting seemed an unlikely cause for commits on the magnitude of 10,000 lines or greater. Upon appeal to the commit log messages, we found a large number of messages containing text such as "Add generated documentation files," "Documentation generated from javadoc," and "Updated documentation using new doxygen configuration."

Further examination revealed that, at least in the cases mentioned above, these commits consist almost entirely of HTML files. The above comments contain 81, 120, and

448 HTML files, respectively. This suggests that large, comment or documentation related commits may be the result of auto-generated HTML files from documentation systems such as javadoc and doxygen.

It is possible that there may be other significant sources of Auto-generated Code expressed in the topics obtained from LDA. In the above case, the tools that generated the "code" were more effective identifiers of Auto-generated code than were the terms "automatically" and "generated." Further investigation is required to determine whether other such cases exist.

4.5.7 Other Findings

As we hoped, the application of LDA to this problem suggested some potential Cliff Wall causes we had not forseen. Additionally, a few interesting observations served to confirm some of our suspicions about Cliff Walls. In this section we discuss some of these findings.

One discovery of note was the importance of activities related to project releases and new versions. The 2nd most prevalent topic discussed in Cliff Wall log messages is "version-release." Topics 11 and 15 in Tbl. 4.2 also deal primarily with project releases and versioning, with prevalent terms such as "latest," "version," "beta," "v1" and "v2." It is difficult to tell exactly what is occurring with these commits; most provide little information other than a version number. We suspect that many of these may be the result of merges, and further investigation may determine the true cause.

We were able to gain some understanding of topics which were infrequently discussed in the log messages of Cliff Wall commits. Table 4.4 shows some of the topics for which the topic rank dropped significantly for Cliff Wall commits. This drop would indicate topics that were discussed much less frequently for Cliff Walls than All Commits, when compared to all other topics. Some of the trends in the table include topics discussing corrective maintenance ("fixed-bug," "fix-typo," 'fixes-minor," "output-debug"), gui tweaks ("button-selection"), and minor implementation details ("string-return," "variable-global," "size-buffer"). It is not

surprising that these topics do not significantly occur in the log messages of large commits, but these trends lend credibility to our results.

Table 4.3 provides another interesting insight. We observe that two topics appear to deal with web technologies: "http-www" and "php-index." In many cases, we found that these topics indicated the presence of a URL in the log message. It is intriguing that this topic surpassed so many others in the Cliff Walls category. We believe that these URLs could convey valuable information about the commit, and may help to identify library code that is being imported, or the location of an external version control repository utilized by the project.

4.6 Threats

Some of the most significant threats to our results arise from the data set employed. As previously stated, the data was gathered in late 2006, and is now relatively old. It is possible that the results that we have found do not correspond to the current state of projects in SourceForge. This study is also limited to projects using the CVS version control system. According to our estimates, almost all new projects in SourceForge are now using Subversion instead of CVS. While the two technologies are similar in many ways, it is possible that our analysis would produce different results if conducted using data from Subversion logs.

It should be noted that when the original data was gathered, projects were filtered to include only those projects listed as "Production," "Stable," or "Maintenance," in an effort to limit the data set to include only "successful" projects [12]. As a result, when we talk about topics across "All Commits," we are actually unable to generalize to the entire population of projects in SourceForge. This is significant, because as one estimate found, only about 12% of projects in SourceForge were being actively developed [34]. It is possible, even likely, that a similar analysis, not limited to "Production/Stable/Maintenance" projects would produce different results. However, we do not feel that the depiction of Cliff Walls would change dramatically, as we presume they are rare in defunct projects.

Another significant threat to the validity of our results is the presence of "low quality" topics. We found two types of low quality topics in our results: topics with contradictory terms and topics generated from dirty data. One example of a topic with contradictory terms is the "removed-deprecated" topic ranked 3rd in Tbl. 4.2. This topic contains the contradictory terms "removed," and "imported" as important terms in the topic. This leads to a topic that is difficult to interpret, as the "meaning" of the topic can vary based on the document in which it is present. To better understand this topic we examined the log messages of 233 Cliff Walls containing the topic. Of those 233, the term "removed" occurred in only 1 message, while "imported" occurred in 154. Obviously, for large commits, "imported" is a much more appropriate description of log messages with this topic.

We found two low quality topics resulting from dirty data in our results. The topics "error-message" and "message-project," the 5th and 6th most prevalent topics for Cliff Walls, are also misleading. We looked at 286 Cliff Wall log messages containing at least one of these two topics, and found that 211 (74%) of them contained only the message "no message." These commits should have been removed from the data set prior to the analysis. Exclusion of these commits would result in a data set containing 2,301,620 commits, with 9,199 Cliff Walls, a minor decrease in size. We do not feel that this issue greatly affected the outcomes of this study. However, these topics possibly prevented more appropriate topics from being considered.

In order to improve the results of future studies applying LDA to the Cliff Walls problem, greater effort should be made in LDA model selection. The three input parameters that we were required to specify (number of topics, number of iterations, and burn-in period) could likely be tuned to produce higher quality topics. In particular, this may help us to avoid the issue of topics with contradictory terms.

4.7 Conclusions

We are excited by the promise that LDA shows for automated analysis of large commits. Through the use of tag clouds and other views of the data, we have been able to gain an insightful picture into the causes behind Cliff Walls. We found that in most cases, our suspicions of Cliff Walls were confirmed. We found significant evidence that library imports, externally developed code, and merges were the subjects of topics frequently discussed in log messages of large commits. We also found evidence that auto-generated code can, in some cases, result in the creation of cliff walls. LDA also helped us to confirm that maintenance tasks, such as bug fixes, do not occur in large commits with much frequency. These conclusions agree with previous studies on the causes of large commits [19, 21].

We found that it was difficult to use commit log messages to distinguish library code imports from imports of large amounts of project code. However, in some cases we are able to identify library imports. We also hope that, in the future, the URLs included in some Cliff Wall commit messages may be used to identify other instances of library code imports.

We believe that LDA is a welcome alternative to many of the methods that have previously been used for classification of commit log messages. While we invested a great deal of time manually interpreting the results produced by LDA, we were able to avoid the tedium of data tagging required by most supervised classification tasks.

4.8 Future Work

The role of large commits in software evolution is still largely unclear. In this study, we have examined the causes of Cliff Walls for a particular subset of all software projects—those that are relatively successful, are hosted on SourceForge, and that use CVS for version control. In order to better understand Cliff Walls, we need to build upon this subset. First, research should consider investigating Cliff Walls as they occur in other version control systems. CVS

is no longer the most significant version control system, since others, such as Subversion and GIT have risen to take its place.

SourceForge is only one of many environments in which open source software is developed. There are various open source forges and foundations, each with its own tools, communities, policies, and practices that influence the software development that occurs therein. It is possible that some of these other environments may prove more welcoming to those interested in studying the evolution of software. An effort should be made to characterize and compare the Cliff Walls that exist in other open source development communities, such as the Apache and Eclipse Foundations, RubyForge, and GITHub, to name a few. Of course the study of large commits should not be limited to only open source organizations, but should be investigated wherever possible.

More information may also be gained through a more in-depth analysis of the Cliff Walls themselves. The largest commit in our data set was over 13 million lines of code. In contrast, the smallest Cliff Wall contained 10,001 LOC. In this study, both of these commits, as well as everything in between, were lumped into the same class: Cliff Walls. It is likely that such a large level of granularity hides much that can be learned about the causes of Cliff Walls. We believe that there are opportunities to better understand this phenomenon by examining more closely the causes behind Cliff Walls of differing magnitudes.

Acknowledgements

The authors would like to thank Dan Walker of the BYU Natural Language Processing Lab for his willingness to provide insight and guidance on the methods used in this paper.

Chapter 5

A Topical Comparison of Monolithic Commits in SourceForge and Apache Foundation Software Projects

Software engineering researchers face many obstacles that complicate the study of software creation. While the use of software "artifacts"—such as mailing list histories and version control logs—can help overcome many of these difficulties, a number of issues prejudice the results of artifact-based studies of software projects. One such threat—the presence of monolithic commits in open source repositories—thwarts attempts to tie authors to contributions, masks true development history, and taints studies of project evolution. In this study we examine the impact of these monolithic commits, which we refer to as "cliff walls", in two open source forges: SourceForge and Apache Foundation. We employ Latent Dirichlet Allocation to infer topics from the log messages of more than 30,000 of the largest commits from SourceForge and Apache Foundation projects. We then analyze these topics to determine the causes of cliff wall commits. We estimate that while cliff walls appear much more frequently in Apache Foundation than SourceForge, the opposite is true when only the most problematic cliff walls are considered. Cliff wall commits occur in both SourceForge and Apache Foundation projects, and thus must be addressed by researchers conducting artifact-based studies on projects from these forges. The Apache Foundation is preferable to SourceForge for certain types of artifact-based studies (particularly those involving software evolution) since the most problematic types of cliff walls are less frequent there.

5.1 Introduction

5.1.1 Artifact-based Software Engineering Research

The software engineering researcher faces numerous obstacles that complicate the study of software creation. Most organizations are loath to allow researchers through their gates, in an effort to protect trade secrets or merely to hide shortcomings. Even in cases where researchers are allowed to directly observe engineers building software, the Hawthorne effect threatens the validity of such observations. Time investment and organizational complexity are also issues that pose problems in software engineering research. Direct observation requires significant time investment, making it impossible for a single researcher to observe everything that takes place within a given software organization.

As a result of these barriers, software researchers take advantage of artifacts—work products left behind in software project burial grounds. Artifacts are collected after the fact, minimizing the confounding influence of the presence of a researcher. Artifacts also help researchers deal with the requirements of studying complex organizations. By leveraging artifacts of the software process, researchers are able to study thousands of pieces of software in a relatively short period of time, an otherwise impossible task.

The open source movement is particularly important to software engineering research, since project artifacts, such as source code, revision control histories, and message boards, are openly available to software archaeologists. This makes open source software an ideal target for researchers with a desire to understand how software is built.

Unfortunately, serious challenges threaten the results of artifact-based research involving open source software projects (such as those hosted on SourceForge and Apache Foundation). Since artifact data is typically isolated from its original development context, it is difficult to ensure that measurements actually measure what they claim to measure [9]. It is necessary, therefore, to validate artifact data, which is generally collected without a targeted purpose.

Many phenomena can easily be overlooked by software researchers, including auto-generated code, the presence of non-source code files in version control, and the side effects of branching and merging. Understanding the limitations of artifact data represents an important step toward validating the results of numerous studies (including, [6, 12, 18, 25, 31, 36, 40]).

This paper addresses one such threat—the presence of monolithic commits in open source software repositories. A close look at the version control history of many projects in SourceForge reveals some worrisome anomalies. Massive commits, which we refer to as "cliff walls," appear with alarming frequency. One investigation of cliff walls in SourceForge found that out of almost 10,000 projects studied, half contained a single commit that represented 30% or more of the entire project size [28]. These cliff walls indicate periods of unexplained acceleration in development, threatening some common assumptions of artifact-based research, especially for studies of project evolution. Cliff walls thwart attempts to tie authors to contributions, mask true development history, and taint studies of project evolution. We must better understand cliff walls if we are to paint an accurate picture of software evolution through the use of artifacts.

5.1.2 Cliff Walls

In [28] we introduced the concept of "cliff walls" to describe unexpected increases in version control activity over short periods of time (see Fig. 5.1). In the most extreme cases, cliff walls represent millions of lines of source code contributed to a repository in a single commit. Such activity is problematic for researchers, especially those investigating the evolution of software, because such sudden surges of source code commits cannot possibly be the result of common incremental development activities. Even a team of "supercoders" would be hard-pressed to produce such massive quantities of code in a short period of time, all the more impossible for the single "author" to which a version control commit is attributed. We are forced to ask what these "cliff walls" truly represent.

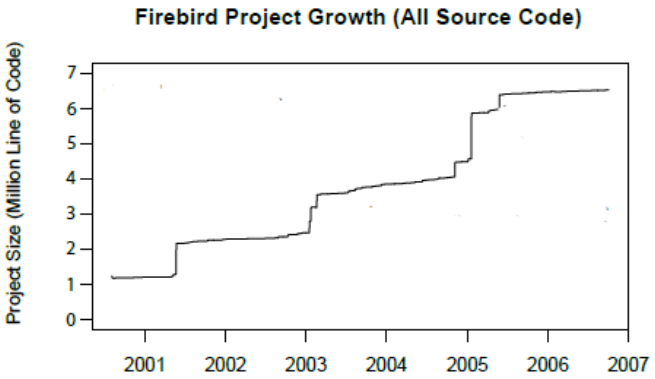


Figure 5.1: Cliff Walls in the Firebird Project

In a prior study we took several steps towards understanding the causes of cliff walls [33]. In that study, we used Latent Dirichlet Allocation (LDA) to extract topics from over 2 million commit log messages, taken from almost 10,000 SourceForge projects. The topics generated through this method were then analyzed to determine the causes of over 9,000 of the largest commits. We found that branch merges, code imports, and auto-generated documentation were significant sources of large commits. We also found that corrective maintenance tasks, such as bug fixes, did not play a significant role in the creation of large commits.

5.1.3 Characteristics of Open Source Forges

Open source software projects reflect tremendous diversity. Most open source software is developed within software forges, which are themselves very diverse. Projects in a forge may congregate around a specific language or technology, as is the case with RubyForge and GitHub. Some forges are focused on developing strong communities around each project, while others seek only to provide tools for open source development. Open source forges even have different rules for participation—while anyone can create a project on SourceForge, the Apache and Eclipse foundations have processes for allowing new projects into their respective forges. In this study, we address several differences between two important open source forges: SourceForge and Apache Foundation.

SourceForge, the largest of all open source forges, is also one of the least restrictive. In many ways, SourceForge represents the "wild west" of open source software development. Project creation within SourceForge is essentially unrestricted—anyone can launch a new SourceForge project with little oversight on the part of SourceForge. The focus of SourceForge is to provide tools needed to run open source projects. Everything else is left up to the project creators, including development of "community" and the creation of software development processes. Because there is so little oversight on many SourceForge projects, the forge is littered with unsuccessful or abandoned projects. A study by Rainer and Gale found that of 50,000 SourceForge projects sampled, only 11.6% were under active development [34].

The Apache Foundation environment differs dramatically from that of SourceForge. Apache is much smaller than SourceForge, and also more strictly controlled. For example, in Apache a well-defined process guides the selection of software projects that become a part of the Foundation. All proposed Apache projects are subject to an "incubation" process, the goals of which include filtering new project proposals, helping with project infrastructure creation, and community growth and supervision. The Apache Foundation also has defined processes for, among other things, project communication, documentation, and decision making [2]. One result of these well-defined processes is that far fewer defunct projects fall under the Apache umbrella. There are only 20 projects listed as retired in the Apache "Attic," suggesting a far greater proportion of projects under active development than is the case in SourceForge [1].

The link between software process and software quality has been studied extensively by software engineering researchers [23]. The software itself is just one of the artifacts left behind by software creation. We suspect that the quality of other artifacts, such as version control and mailing lists histories is similarly affected by software process. For example, it is highly likely that mailing list archives will be more complete in Apache Foundation projects—where process dictates that developers utilize mailing lists—than mailing list archives on SourceForge projects, where no such dictate exists.

In this study we examine the role that forge personality plays in the quality of artifacts left behind by a community of developers, specifically in the case of version control logs. We originally hypothesized that, due to the well-defined processes enforced by Apache Foundation, 1) cliff walls would occur less frequently in Apache Foundation projects than in SourceForge projects, and 2) the most problematic types of cliff walls would occur with greater frequency in SourceForge projects than in Apache Foundation projects. As we illustrate in Section 5.5, our first hypothesis is found to be incorrect, while we find significant evidence supporting our second hypothesis. These results suggest that the Apache Foundation is preferable to SourceForge for certain types of artifact-based studies—particularly those that study software evolution.

5.2 Definitions

Table 5.1: Examples of Duplicate "Initial revision" commits in SourceForge

Project	Timestamp	Username	Log message	Commit size
anarchism	7/30/2000 16:59	mdillon	importing HTML.	71,824
anarchism	7/30/2000 16:59	mdillon	Initial revision	71,824
cayenne	3/18/2002 23:15	mensk	Initial revision	40,703
cayenne	3/18/2002 23:15	mensk	importing Cayenne to SourceForge. original CVS tag was sf-migrate	40,703
patchhack	3/22/2002 23:56	jdorje	Initial revision	240,706
patchhack	3/22/2002 23:56	jdorje	Imported initial 3.4.0 sources.	240,706
dmx	1/26/2004 21:14	faith	Initial revision	174,307
dmx	1/26/2004 21:14	faith	XFree86 CVS Repository Import (using xf-4_3_99_902 tag)	174,307

Contributions to version control systems are typically examined in the context of "commits." However, some researchers have defined "commit size" in terms of the number of files changed in a commit [19–21], while others treat the size of a commit as a function of the number of LOC added, removed or modified [4].

Within the context of this study, a *commit* refers to the set of files and/or changes to files submitted simultaneously to a version control system by an individual developer. Since we are primarily concerned with the growth of projects over time, *Commit size* is defined as the total number of lines added or removed from all source code files in a given commit, which allows us to perceive code growth (or shrinkage) within a project. For example, if an

existing line in one file is modified, and no other changes are made, the size of the resulting commit would be zero LOC. In this study, we include code comments in the measurement of commit size, allowing us to detect code growth attributable to the insertion of comments within source files.

We use the term *Cliff Wall* to describe any single commit with size greater than 10,000 lines of code (LOC). We believe this threshold to be sufficiently high that it avoids capturing most incremental development contributions for individual authors. While it is possible that an extremely productive developer could produce 10,000 lines of code in a period of days or weeks, it is unlikely. Additionally, the methods employed in this study should allow us to identify instances where the threshold fails.

5.3 Latent Dirichlet Allocation

Latent Dirichlet Allocation (LDA) is an unsupervised, hierarchical Bayesian topic model for analyzing natural language document content that clusters the terms within a corpus into topics [8].[1] In LDA, the model represents the assumption that documents are a mixture of these topics and that each topic is a probability distribution over words. In short, LDA allows us, without any knowledge engineering, to discover the topical content of a large collection of documents in an unsupervised fashion. Given a corpus of documents such as our commit log messages, inference techniques such as Gibbs sampling or Variational Bayes can be used to find a set of topics that occur in that corpus, along with a topic assignment to each word token in the corpus.

A common technique for visualizing the topics found using LDA is to list the top n most probable terms in each topic. For example, one of the topics we found using LDA that concerns bug correction, consisted of the following terms:

fix bug fixes typo problem warning

compile problems small

[1]The discussion of LDA in this section is borrowed from our discussion in [33].

When brevity is required, it is also common to refer to a topic by just the first two or three most probable terms, sometimes in hyphenated form. Using this notation, the topic above could be referred to as the "fix-bug" topic. Both of these methods for indicating a particular topic (word list and first words) are used throughout this paper.

Because LDA also assigns a topic to each individual token within a commit message, a single commit can, and usually does, contain many topics. This is one benefit of using LDA: each "document" may belong to multiple classes. Such is not the case with many of the supervised classification methods, which typically assign each document to a single class. In the case of commit log messages, allowing for multiple topics in a single message allows us to conduct a more fined-grained analysis.

The approach taken by LDA is an attractive alternative to other classification and clustering methods. Supervised classification methods require a training set of "tagged" data, for which the appropriate class for each instance has previously been specified. In many situations this tagged data does not exist, requiring the researcher to manually assign tags to some portion of the data. This tedious, time-consuming, and potentially biased process can be avoided through the use of unsupervised methods such as LDA. Unsupervised methods may also compensate for a measure of "short-sightedness" by the researcher. In supervised methods, since the classes must be predefined by the researcher, it is possible to degrade the usefulness of the model through the omission of important classes, or the inclusion of irrelevant classes. Unsupervised methods avoid some of these errors since no predefined clusters are specified, allowing the data to better "speak for itself." For our analysis, we used the open source tool "MALLET" which implements LDA [29].

5.4 Data & Analysis

In order to make a comparison between Apache and SourceForge we needed to gather data from both forges. The following section provides a description of the data. A summary of the composition of the data can be found in Table 5.2.

5.4.1 SourceForge

The SourceForge data included in this study was gathered in late 2006, and includes the CVS version history of almost 10,000 projects. We previously used this data set in several studies [12–14, 25–28, 33]. Because CVS logs do not maintain any concept of an atomic multi-file "commit" it was necessary to infer individual commits. To do this we utilized the "Sliding Time Window" method introduced by Zimmerman and Weißgerber [41]. The resulting data set consisted of 2,489,561 commits from 9,671 projects. Of those, 15,579 commits were identified as cliff walls.

A close inspection of the SourceForge commits revealed that some were redundant and needed to be filtered before our analysis could be conducted. Certain conditions, such as the use of the CVS "import" command, result in the creation of duplicate file revisions in a CVS repository. The duplicate revisions carry the log message "Initial revision", and if left in the data, inflate the number of commits, especially in the case of cliff walls. Table 5.1 shows several examples of these duplicate commits. After the removal of these spurious commits, the SourceForge data consists of 2,465,374 commits, including 10,843 cliff walls. The bulk of the data spans from 1996 through October 2006; Only 43 of the projects in the SourceForge data contain commits that pre-date 1996.

5.4.2 Apache Foundation

The Apache Foundation data was gathered much more recently, and runs from its inception in 1996 through January of 2012. The Apache Foundation currently uses a single Subversion repository to host all of its projects, making it difficult to get an exact count of the number of projects represented in the Apache data. We were able to estimate the number of projects by simply counting the number of projects listed on the Apache Foundation website. We included projects listed as a part of the Apache Incubator and those currently in the Attic, yielding a total of 226 Apache projects [1, 3]. Since Subversion tracks individual commits, the cumbersome process of inferring commits from CVS (required in our gathering of SourceForge

data) was unnecessary for Apache. The Apache data contains 1,226,355 commits, including 19,761 cliff walls.

As we did for the SourceForge data, we checked the Apache data for spurious "Initial revision" commits. While we did encounter several commits with this log message, we were unable to find the corresponding duplicate commits that would warrant the removal of "Initial revision" commits in Apache.

5.4.3 Commit Message Topic Composition

As a first step in examining the commit message data, we compared the topical composition of various subsets of the data by computing the Kullback-Leibler (KL) divergence between these subsets. KL divergence is an asymetric measure of similarity between two distributions which is equal to 0 if and only if the two distributions are equal.

We learn two important things from our calculations of KL divergence, the results of which are displayed in Figure 5.2.[2] First, we learn that the topic distribution for cliff walls in Apache Foundation is significantly different from the topic distribution for cliff walls in SourceForge. This knowledge provides confidence that differences between Apache Foundation cliff walls and Source Forge cliff walls do in fact exist. Second, we see that there are significant differences in the topic distribution for cliff walls in Apache Foundation commits from 2006 and earlier (Apache Interval 1) when compared to Apache Foundation commits occurring after 2006 (Apache Interval 2). These results confirm the suspicion that there may exist a temporal influence that had the potential to confound our analysis. To avoid such influence, we limit the bulk of our comparison to SourceForge and the Apache Interval 1 data, since these portions of the data cover roughly the same time span. This comparison is robust to temporal factors.

[2]The color shade in this graph is a an indicator of the magnitude of the difference in topic distribution between two classes. A deep blue shade indicates no or very little difference. As the color progresses towards the red end of the spectrum the difference between topic distributions grows.

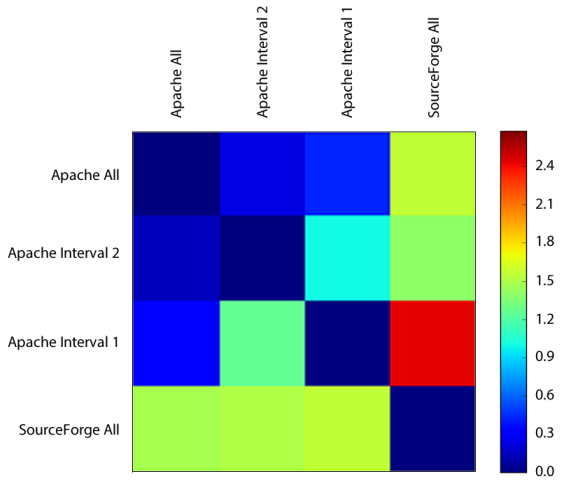


Figure 5.2: KL Divergence between Cliff Wall classes

Table 5.2: Summary of the Data

	SourceForge	Apache		
		Interval 1	Interval 2	Total
# Projects	9,671	-	-	≈ 226
Commits	2,465,374	549,231	677,123	1,226,355
Cliff Walls	10,843	7,283	12,478	19,761
Per Project	1.1212	-	-	≈ 87.4381
Per Commit	0.0044	0.0133	0.0184	0.0161

5.4.4 Analysis

Our approach to measuring the relative impact of cliff walls in SourceForge and the Apache Foundation consists of two primary parts. First, we gathered some simple statistics on the frequency with which cliff walls occur in each forge. Second, as certain types of cliff walls have a more negative effect than others, we applied LDA to identify the different types of cliff walls that occur in these forges, and then compared the frequency of these between the two forges.

In order to ensure that the topics generated by LDA were consistent, we combined the Apache and SourceForge data when running LDA. The LDA algorithm, as implemented in MALLET, requires three input parameters: 1) the number of topics to produce in its analysis, 2) the number of iterations, and 3) the length of the burn-in period. In our study, we elected

to identify 100 topics with MALLET. Hindle and German identified 28 "types of change" for the commits classified as a part of their taxonomy [21]. Hattori and Lanza, in their study of commit messages, identified 64 "keywords" that were used to classify commits [19]. These prior results gave us reason to believe that 100 topics would be a sufficient number to capture the motivations behind the commits in the data set, with an appropriate level of detail.[3]

After obtaining topic distributions for each commit, we manually analyze these topics and determine the impact of each for cliff walls within each forge. Due to the time requirements for such a manual analysis, instead of examining all 100 topics, we instead focus our analysis on the top 10 most prevalent topics for each segment of the data. In doing so, we also focus on the types of cliff walls that likely have the largest impact.

5.5 Results

5.5.1 Frequency of Cliff Walls

We first examined the frequency of cliff walls within Apache Foundation and SourceForge projects. Monolithic commits appear much more frequently in Apache Foundation projects than they do in SourceForge projects, as can be seen in Table 5.2. The table shows that cliff walls occur over 3 times more often per commit in Apache Foundation than they do in SourceForge. Even worse, the average Apache project is estimated to contain almost 80 times as many cliff walls as the average SourceForge project. This result clearly flies in the face of our original hypothesis. A number of factors may contribute to this discrepancy, as we discuss in detail in Section 5.6.1.

This is not the entire story, however. As previously described, certain types of cliff walls have greater impact on artifact-based research than do others. In many cases, cliff walls can simply be excluded from an analysis with no further impact on results. In the following

[3]The other two parameters, number of iterations and length of burn-in period, are required by the Gibbs Sampling inference method employed by MALLET. We refer the reader to [39] for a description of LDA as it is implemented within MALLET, including a description of Gibbs Sampling. For these two parameters we used the default values provided by MALLET; 1,000 iterations with 200 dedicated to burn-in.

Table 5.3: Uninformative Log Messages

	Per Cliff Wall			Per Commit		
SourceForge	1,614	/ 10,843	(14.89%)	1,614	/ 2,465,374	(0.07%)
Apache	393	/ 19,761	(1.99%)	393	/ 1,226,355	(0.03%)
Interval 1:	168	/ 7,283	(2.31%)	168	/ 549,231	(0.03%)
Interval 2:	225	/ 12,487	(1.80%)	225	/ 677,124	(0.03%)

sections we identify the most problematic types cliff walls and attempt to gauge their impact within the projects of interest.

5.5.2 Uninformative Log Messages

Because LDA utilizes commit log messages to determine the type of cliff wall, empty or other uninformative log messages pose a problem to this type of analysis. LDA is unable to generate meaningful topics for these commits. Uninformative log messages were common in both forges, as can be seen in Table 5.3. These types of cliff walls appear more than twice as often per commit in SourceForge (0.07%) as in Apache Foundation (0.03%). It is also interesting to note that almost 15% of SourceForge cliff wall log messages contain no information that would help in determining the cause for such a large commit. The corresponding values in Apache foundation are much smaller (1.80 - 2.31%).

It is important to note that we were not exhaustive in our analysis of what might be considered an "uninformative log message". The log messages "empty log message", "no message", and "no log message" were common examples of uninformative messages, and all instances of these were included in our estimate. Additionally, we used a regular expression to find blank entries or especially problematic log messages, such as messages containing only special characters. [4]

[4]We used the POSIX regular expression **^[^A-Za-z0-9]*$** to find uninformative log messages in our Postgresql database.

5.5.3 Topic Analysis of Cliff Walls

In this section we discuss some of the most prevalent topics for cliff walls in both SourceForge and Apache Foundation projects. Tables 5.4, 5.5 & 5.6 each contain a list of the top 10 topics for each portion of the data, as well as the percentage of tokens that were assigned to that topic by the LDA algorithm. For instance, the most prevalent topic for SourceForge cliff walls (see Table 5.4), was the "initial-revision" topic. The table shows that 23.75% of all tokens in SourceForge cliff walls were assigned to this topic. We limit the majority of the discussion to SourceForge and Apache "Interval 1", as the comparison between these two groups is the most relevant.

As previously mentioned, we focus our analysis on the top 10 topics from each segment of the data. Tables 5.4, 5.5, & 5.6 indicate that the top 10 topics account for 51.92% of all words in SourceForge cliff walls, 75.66% of all words in Apache Interval 1 cliff walls, and 67.59% of all words in Apache Interval 2 cliff walls. In other words, more than 50% of the words in cliff wall log messages are assigned to the top 10% of topics. Since several of the top 10 topics are shared across the three different portions of the data, we analyzed a total of 17 unique topics.

"initial-revision"

The "initial-revision" topic is one of the most important in our analysis, for two reasons. First, it is quite prevalent for cliff walls, appearing in the top 10 topics for both SourceForge and Apache "Interval 1". Second, we believe this topic to be a strong indicator of a negative side effect of cliff walls—the loss of development history. In almost all cases this topic is assigned to commits that represent the initial addition of files to the project repository. Cliff wall commits containing this topic are accompanied with log messages similar to the following:

> "Initial commit", "Initial revision", and "Initial CVS import"

These imports fall into one of two categories: External Code Imports or Project Code Imports. In the first case, code that was not developed as a part of the project is imported into the repository. The most common occurrence of this type of import involves software library source code added to the repository. The second type, project code import, occurs when code developed as a part of the project of interest is added to the repository in large quantities. When large amounts of code are added in this manner, granular development history of the code is lost, rendering analysis excessively difficult.

"message-log"

The second most prevalent topic in SourceForge is "message-log", the direct result of log messages containing uninformative text. 1,327/1380 (96.2%) of SourceForge cliff walls with at least one word assigned to this topic consisted of either the log message "empty log message" or "no message". Commits of this type were addressed in Section 5.5.2.

"version-number"

The topic "version-number" also appeared in the top 10 topics for both SourceForge and Apache Interval 1, however it seemed to represent slightly different things in the two forges. While messages in both forges reference new software versions or releases, the log messages in Apache also often specifically mention the creation of branches or tags in many of these messages. For example:

> "Tagging 1.4alpha1","Branching for 0.3 releases." and "Tagging Tomcat version TOMCAT_5_5_20."

These messages are representative of Apache commits with this topic, while in SourceForge branches and tags are rarely mentioned:

> "Beta version 1.0.001","Updated to 2.4.0-test9-pre2" and "Updated to 1.1rc1"

Table 5.4: Top 10 Topics for SourceForge

#	%	Topic Terms
1	23.75%	**initial revision** import padding checkin
2	10.73%	**message log** empty zadny autoresponse
3	3.48%	**version number** updated changed versions
4	2.50%	**api add** adding update repository
5	2.46%	**moved package** module move code
6	2.34%	**update updated** documentation version docs
7	1.95%	**release plugin** maven prepare tag
8	1.62%	**code cleanup** removed clean cleaned
9	1.60%	**project xml** pom files adding
10	1.49%	**added support** multiple files ability
Total	51.92%	

This discrepancy may be due to differences between the CVS and SVN version control systems, as it is unlikely that Tag or Branch Creation will result in a cliff wall in CVS. This issue is discussed more fully in Section 5.6.1.

"api-add"

The topic "api-add" makes it into the top ten only for SourceForge cliff walls. Many of the log messages for this topic represent initial import activities, much like the "initial-revision" topic. The log message "Imported sources" is very common for this topic. Some messages for this topic indicate the addition of library code to the project repository. For instance, one commit message stated "Imported zlib 1.1.4". A brief web search revealed that zlib is a compression library. In other cases, these messages were more indicative of the addition of code developed as a part of the project itself. One example was taken from the SourceForge project zodiac, the log message reading "Imported from zodiac-0.6.5".

Table 5.5: Top 10 Topics for Apache Interval 1

#	%	Topic Terms
1	48.82%	**commit tag** create cvs2svn manufactured
2	11.17%	**release plugin** maven prepare tag
3	3.66%	**version number** updated changed versions
4	3.02%	**initial revision** import padding checkin
5	2.19%	**merge trunk** svn branch status
6	1.67%	**file branch** added initially java
7	1.51%	**build jar** xml ant geronimo
8	1.41%	**project xml** pom files adding
9	1.29%	**update updated** documentation version docs
10	1.21%	**file directory** files path dir
Total	75.66%	

Table 5.6: Top 10 Topics for Apache Interval 2

#	%	Topic Terms
1	32.87%	**release plugin** maven prepare tag
2	12.99%	**fixes minor** change updates small
3	4.68%	**version number** updated changed versions
4	4.08%	**merge trunk** svn branch status
5	3.85%	**commit tag** create cvs2svn manufactured
6	2.43%	**file branch** added initially java
7	2.00%	**project xml** pom files adding
8	1.69%	**fix issue** merged head wicket
9	1.65%	**file directory** files path dir
10	1.38%	**update updated** documentation version docs
Total	67.59%	

moved-package

The topic "moved-package" was also unique to the SourceForge top 10. Cliff walls with this topic largely represent actions related to repository restructuring, such as rearranging

directory layout. It is not surprising that such actions would result in cliff walls, as any copied files would need to be "added" to the repository in a new location.

update-updated

The topic "update-updated" was included in the top 10 for SourceForge and both Apache intervals. It is evident from log messages that this topic often represents updates to project documentation in both Apache and SourceForge. In the case of Apache projects, it was also common to see references to "website" updates:

"Updated site with latest changes.", "Update web site" and "Updated jaxme website"

These messages may indicate updates to the Apache Foundation website itself, as the website code is maintained in the Apache Subversion repository.

release-plugin

The "release-plugin" topic is similar to the "version-number" topic for both Apache and SourceForge cliff walls. Much like "version-number" this topic is indicative of activities related to software releases. In the case of Apache, almost all of these cliff walls are Subversion tags. It appears that these commits are the result of "tagging" a version of the code as a part of a code release.

commit-tag

The "commit-tag" topic appeared in the top 10 only for Apache projects. Like several other important SourceForge topics, commits containing this topic are almost exclusively the result of Subversion tags. In Subversion, a tag is simply a "snapshot" of the repository at a given time—these commits contain no real development history. Cliff walls that are the result of Subversion tags can simply be excluded in most cases with no negative side effects.

merge-trunk, file-branch and file-directory

The "merge-trunk", "file-branch", and "file-directory" topics also appear in the top 10 exclusively for Apache. Log messages indicate that these cliff walls were the result of activities performed on branches—specifically the creation of new branches and the merging of branches. These activities are likely to result in the creation of cliff walls, but much like tag creation, do not represent a significant threat to many artifact-based studies.

fixes-minor

While this topic appears only in the top 10 for Apache Interval #2, and thus isn't a crucial part of our comparison, we feel that it needs to be addressed thoroughly, as it is one of the more confusing topics to make an appearance in the top 10. The top terms in this topic are indicative of bug fixes or other relatively small changes. It is unlikely that such activities would generate the 10,000 lines of code necessary to categorize such a commit as a cliff wall. On closer examination, we found that the message "promotions validated" constituted 1,544/1,758 (87.8%) of cliff walls commits containing the "fixes-minor" topic. To further understand what was occuring in these cases, we manually examined the paths of the files that were included in these commits. In all cases, these appear to be Subversion tags created as a part of the Apache SpamAssasin project.

Interestingly, we found a few clues explaining the discrepancy between the topic and the log messages assigned to it. By searching for log messages containing the words "promotion" or "promotions" we encountered the following:

> "Suicide support bugfix in promotions", "small fix to get promotions working again" and "Small fix to check for unapply on expired promotions, ie only filter by date if this is an apply operation."

These log messages suggest a relationship between promotions and minor changes that LDA would likely identify, resulting in the situation described above. This particular

instance illustrates some of the shortcomings of the methods we apply in this study. The topic "fixes-minor" gives us little explanation for the actual cause of the cliff wall commits with this topic. While this is by far the most troublesome topic we encountered, such occurrences lead us to consider improvements to our methodology. Some of these ideas are discussed further in Section 5.7.

5.5.4 Interpretation Difficulty

Some LDA topics are more difficult to interpret than others. Several of the top 10 topics in the data set fall under such a description: "code-cleanup", "project-xml", "build-jar", and "fix-issue". We found that when we examined log messages for these topics there was no single concept that described the majority of log messages for that topic. In these cases, log messages are dependent on multiple topics for interpretation. To further illustrate this issue, we examine two of the top 10 topics more closely: "Message-log" and "Project-xml" from SourceForge.

The "message-log" topic was one of the easiest to interpret. Almost all cliff wall messages with this topic contained only the uninformative message "empty log message." Commits containing the "project-xml" topic, on the other hand, were much more difficult to interpret. Some messages from these topics indicated initial imports, others repository restructuring and still others updates to project documentation:

> "Moving sources to a src subfolder", "adding new javadoc", "Adding Xcode project file." "First import of sources", and "Lots of changes"

To further demonstrate the differences between these topics, we tabulated the number of topics assigned to each commit. Figure 5.3 contains the resulting histogram for these two topics, with the median overlayed in blue and the mean in red. We can see that for "message-log", the median is 1, indicating that at least 50% of the commits that contain this topic contain *only* this topic. In contrast, a very small number of commits that contain the

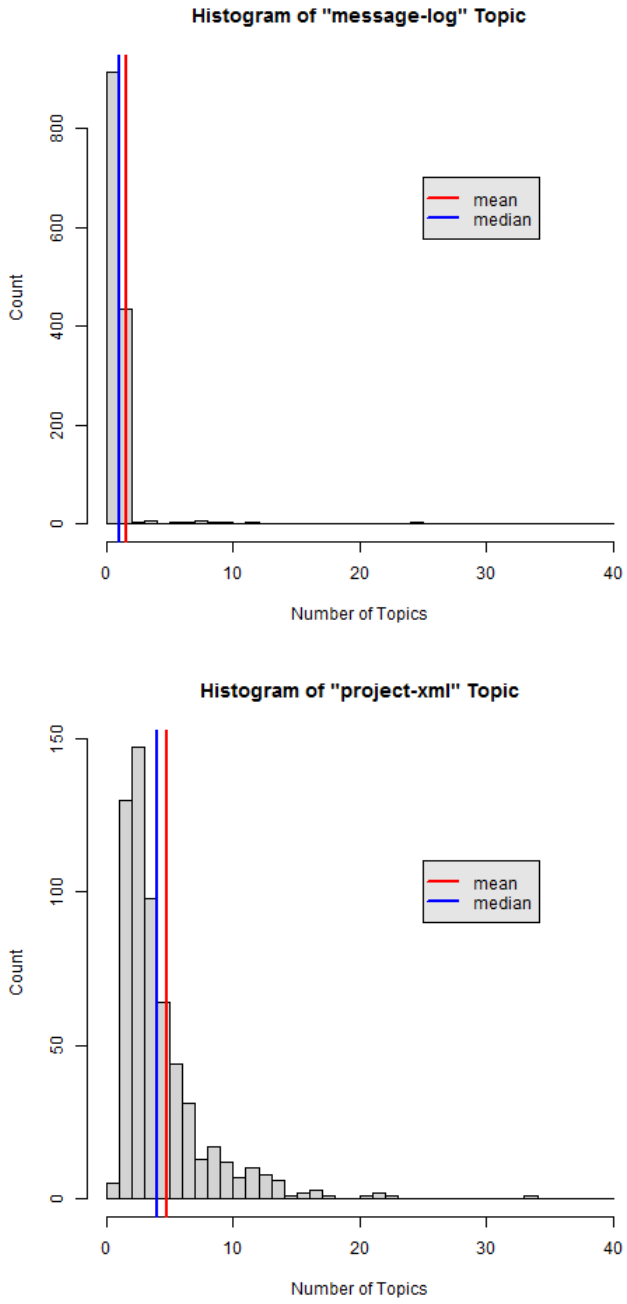


Figure 5.3: Topics per Log Message: "Message-log" & "Project-xml"

"project-xml" topic contain exclusively that topic. It is logical to conclude that the closer a given topic's median is to 1, the easier it will be to interpret associated commit messages, without the need to examine additional topics.

This highlights another difficulty in our analysis. Future work must involve greater efforts to understand topics as they co-occur.

Table 5.7: Prevalence of Problematic Cliff Walls

	SourceForge			Apache Interval 1		
	# Commits	Per Commit %	Per Cliff Wall %	# Commits	Per Commit %	Per Cliff Wall %
Uninformative	1614	0.07%	14.89%	168	0.03%	2.31%
Initial-revision	2861	0.12%	26.39%	200	0.04%	2.75%
Api-add	262	0.01%	2.42%	27	<0.01%	0.37%
Total	4737	0.19%	43.69%	395	0.07%	5.42%

5.5.5 Problematic Cliff Walls

In the previous section, we identified the two classes of cliff walls that pose the greatest threat to studies of version control histories: 1) Cliff walls with uninformative log messages and 2) Cliff walls that represent a loss of granular development history. Through our analysis, we were able to identify examples of both.

In Section 5.5.2, we distinguished the commits in our data set that fall under the first category. We have pinpointed two topics that are useful in identifying commits that belong to the second category of harmful cliff walls: "Initial-revision" (see Section 5.5.3) and "Api-add" (see Section 5.5.3). With the data generated by LDA, we are able to identify each cliff wall for which a portion of the log message is attributed to these topics.

Table 5.7 contains a summary of the impact of these cliff walls in SourceForge and the corresponding portion of the Apache data set. We estimate that about 1 out of every 520 commits in SourceForge is a problematic cliff wall. For comparison, in Apache Interval 1, this ratio is closer to 1 out of every 1390 commits[5]. Note that in Section 5.5.1, we found that cliff walls occurred more frequently in Apache Foundation than in SourceForge. We see here, however, that this relationship is inverted when we examine only the most problematic cliff walls—such commits are *less* frequent in Apache Foundation than in SourceForge.

[5]It is important to note that in order for a commit to be counted as "belonging" to either the "initial-revison" or "api-add" topics, we required that at least half of the words in the log message be attributed to that topic.

5.6 Conclusion

While cliff walls occur with much greater frequency in Apache Foundation than in SourceForge (see Section 5.5.1), we estimate that the most harmful types of cliff walls occur more than twice as often in SourceForge than in Apache Interval 1 (see Section 5.5.5). Unfortunately, our methods have shortcomings that limit the strength of our conclusions. In the following sections we articulate the most significant insights gained from our research.

5.6.1 Branching and Tagging

In this study we compare two important open source forges: Apache Foundation and SourceForge. The data gathered from each of these forges was stored in different version control systems—the Apache Foundation utilizes Subversion while, at the time the data was gathered, SourceForge projects used CVS. While these version control systems have much in common (especially when compared to their distributed "cousins") their differences make comparison difficult in some cases. One significant difference between CVS and SVN is how these systems implement branches and tags.

In both SVN and CVS a tag is intended to provide a "snapshot" of the repository. Tags are used in situations where it may be desirable to easily retrieve the state of the repository for a specific moment in time. This functionality is commonly used within a project in preparation for the release of a new version of the software. The code base is tagged so that an exact copy of the release code can easily be retrieved in the future. Tags, though conceptually similar in both CVS and SVN, are implemented differently in these two systems. When code is tagged in Subversion, a copy of the tagged code is stored in the repository, often in a special "tags" directory. This often results in a large commit—each file is added to the repository as if it were a new file, since it is "new" in this location. In CVS, however, no such copy is made, and no new revision numbers are assigned to files. As a result, it is quite common for tag creation in Subversion to result in a cliff wall, while the act of creating a tag in CVS will never result in a cliff wall.

Branches, like tags, are also similar in CVS and SVN at a conceptual level. In both systems, branches allow for efficient concurrent development. However, when a branch is created in Subversion, a copy of the branched code is made. Changes made on a branch can later be re-incorporated with the trunk through a merge. Both branch creation and merging can result in cliff walls in Subversion. Branch creation, when viewed in the Subversion logs, appears identical to tag creation—a copy that reflects the addition of many "new" files. When a merge occurs, the entire history of changes made on a branch are grouped into a single commit. If significant development has taken place on the branch since it was last merged it is possible that the resulting commit would be greater than 10,000 LOC—classifying the commit a cliff wall. In CVS no copy is made when a new branch is created, so no cliff wall appears. A merge in CVS, though, could result in a cliff wall. A summary of these differences is expressed in Table 5.8.

Table 5.8: Possibility of cliff wall

	SVN	CVS
Branch Creation	T	F
Tag Creation	T	F
Branch Merge	T	T

In Section 5.5.1, we found that cliff walls were much more prevalent in Apache Foundation projects than in SourceForge projects, both on a per commit and on a per project basis. It is likely, however, that the estimate of Apache Foundation cliff walls is inflated by the presence of many "Tag" and "Branch" cliff walls. In Section 5.5.3 we noted that several of the top 10 cliff wall topics in our Apache data indicated branch and tag creation. Because these types of cliff walls do not appear in projects using CVS, our estimate of cliff walls in Apache Foundation is likely inflated relative to our estimate of Apache Foundation cliff walls. Further work should be conducted to determine the magnitude of the overestimate. It is important to note that this issue affects only the raw cliff wall counts—our manual analysis of problematic cliff walls (see Section 5.5.3) does not suffer from this problem.

5.6.2 "Cross-contamination"

Table 5.9: Evidence of Repository Conversion in the Apache Foundation

Revision #	Username	Timestamp	Log message
502855		8/20/1994 19:16	New repository initialized by cvs2svn.
502856	fielding	8/20/1994 19:16	**Initial revision**
...			
76309		1/14/1996 11:49	New repository initialized by cvs2svn.
76310	cvs	1/14/1996 11:49	**Initial revision**
76311		1/14/1996 11:49	This commit was manufactured by cvs2svn to create branch 'unlabeled-1.1.1'.
76312	cvs	1/14/1996 11:49	**Initial revision**
76313	cvs	1/14/1996 11:49	**Initial revision**
76314	cvs	1/14/1996 11:49	**Initial revision**
76315	cvs	1/14/1996 11:49	**Initial revision**
76316		1/14/1996 11:49	This commit was manufactured by cvs2svn to create branch 'APACHE_1_0_0'.
76317	cvs	1/14/1996 11:49	Import Apache 1.0.0
76318		1/14/1996 11:49	This commit was manufactured by cvs2svn to create tag 'apache_1_0_0'.
76319		1/14/1996 11:49	This commit was manufactured by cvs2svn to create branch 'apache_1_0_1'.
...			
77758	dgaudet	3/24/1997 21:43	**Initial revision**
77757		3/24/1997 21:43	New repository initialized by cvs2svn.
77759		3/24/1997 21:43	This commit was manufactured by cvs2svn to create branch 'apache'.
52379		3/24/1997 21:43	New repository initialized by cvs2svn.
52380	dgaudet	3/24/1997 21:43	**Initial revision**
52381		3/24/1997 21:43	This commit was manufactured by cvs2svn to create branch 'apache'.

The difficulties articulated in the previous section make comparison between project histories from different version control systems more difficult. It would seem, however, that a simple solution exists: avoid comparisons between project histories from different version control systems. This is more difficult to accomplish than it originally appears.

The software industry is rapidly evolving. New technologies constantly emerge, only to be replaced in a few years. This holds true for the tools that facilitate the creation of software, such as version control systems. CVS was largely supplanted by Subversion, which in turn is being replaced by tools such as GIT. In many cases, long running software projects are compelled to keep up with these trends, perpetually migrating their project history from one system to another.

When the Apache Foundation was formed in 1999, development for Subversion had not even begun. It was therefore impossible for the earliest Apache Foundation projects to have utilized Subversion. Hence, we see evidence of Apache's transition to Subversion in our analysis. Many of the Apache cliff walls belonging to the "commit-tag" topic make reference to cvs2svn, an open source tool for migrating version history from a CVS repository to other

version control systems [10]. When examining "Initial revision" commits in SourceForge, we found that the occurrence of this log message can be attributed to a cvs2svn import. Table 5.9, which contains excerpts from the Apache Subversion logs in chronological order, demonstrates this fact.

This demonstrates that version control histories of long running projects may contain relics of a conversion from an older system. These relics pose another threat to artifact-based research of version control systems, though determining the magnitude of this threat is outside of the scope of this study. Researchers must be wary of such relics, even when comparing or analyzing project histories hosted in the same version control system.

5.6.3 Benefits of Mining SVN

Another difference between version control systems is the degree of historical data they record. We have found that one version control system may provide the software archaeologist with useful information that is absent in other version control systems. This is the case with SVN when compared to CVS. Subversion provides a number of pieces of information of which software researchers may be able to take advantage. For example, SVN tracks atomic commits, while CVS does not. While solutions to this problem exist, the additional processing required when analyzing CVS makes it a less attractive target for data miners. Additionally, Subversion provides information about the type of changes that occur (a feature absent in CVS). In Subversion commit logs, each file or directory entry is accompanied with a change type that falls into one of 4 categories: Added, Deleted, Modified, or Replaced. In the case of copied files or directories, Subversion logs also indicate the original location of a file or directory. Methods for inferring information from CVS are difficult and time consuming.

5.7 Future Work

We are pleased with the insight that LDA has provided concerning the cliff wall phenomenon—we now have a better idea of the actions that typically result in the creation of cliff walls.

While we are able to identify global trends for cliff walls, we do not feel that our current methods are yet sufficient for accomplishing our ultimate goal—automated identification and resolution of individual cliff walls. In order to achieve these goals, we must be able to classify and handle individual cliff walls with greater confidence.

As future work towards this goal, we propose an approach more dependent on machine learning techniques. We feel that a supervised approach, with manually classified cliff walls for model training and test, would allow us to identify causes for individual cliff walls with greater accuracy. LDA provides a rich set of features that could be used to train an automatic classifier: per commit topic proportions, number of topics per commit, and even topic dyads or triads. In addition to LDA features, a supervised machine learning approach could take advantage of other information gathered from the version control systems themselves. For example, file path information and change type, where available, would provide extremely valuable data for an automatic cliff wall classifier. These sources of information may also prove useful in situations where LDA is unable to provide insight, such as in the case of blank or uninformative log messages.

Our hope is that such a classifier would also better distinguish between External Code and Project Code imports (an extremely important distinction in determining how to properly handle cliff walls). We are highly optimistic that future investment in this area could one day lead to automated tools capable of measuring and improving the quality of software artifacts gathered from version control systems.

References

[1] Apache Software Foundation. The Apache attic, 2011. Accessed, November 2012. <http://attic.apache.org>.

[2] Apache Software Foundation. How the ASF works, 2012. Accessed, November 2012. <http://apache.org/foundation/how-it-works.html>.

[3] Apache Software Foundation. Apache Software Foundation index: Alphabetical index, 2013. Accessed, November 2012. <http://projects.apache.org/indexes/alpha.html>.

[4] O. Arafat and D. Riehle. The commit size distribution of open source software. In *Proceedings of the Hawaii International Conference on System Sciences*, 2009.

[5] P. Bahn, P. Bahn, and B. Tidy. *Archaeology: a very short introduction*. Oxford University Press, USA, 2000.

[6] C. Bird, A. Gourley, P. Devanbu, A. Swaminathan, and G. Hsu. Open borders? immigration in open source projects. In *Proceedings of the International Workshop on Mining Software Repositories*, 2007.

[7] C. Bird, D. Pattison, R. D'Souza, V. Filkov, and P. Devanbu. Latent social structure in open source projects. In *Proceedings of the International Symposium on Foundations of Software Engineering*, 2008.

[8] D. Blei, A. Ng, and M. Jordan. Latent dirichlet allocation. *The Journal of Machine Learning Research*, 3:993–1022, 2003.

[9] L. C. Briand, S. Morasca, and V. R. Basili. Defining and validating measures for object-based high-level design. *IEEE Transactions on Software Engineering*, 25(5):722–743, 1999.

[10] CollabNet. cvs2svn project home, 2009. Accessed, November 2012. <http://cvs2svn.tigris.org/>.

[11] C. de Souza, J. Froehlich, and P. Dourish. Seeking the source: Software source code as a social and technical artifact. In *Proceedings of the International Conference on Supporting Group Work*, 2005.

[12] D. P. Delorey, C. D. Knutson, and S. Chun. Do programming languages affect productivity? A case study using data from open source projects. In *Proceedings of the International Workshop on Emerging Trends in FLOSS Research and Development*, 2007.

[13] D. P. Delorey, C. D. Knutson, and C. Giraud-Carrier. Programming language trends in open source development: An evaluation using data from all production phase sourceforge projects. In *Proceedings of the International Workshop on Public Data about Software Development*, 2007.

[14] D. P. Delorey, C. D. Knutson, and A. MacLean. Studying production phase SourceForge projects: A case study using cvs2mysql and sfra+. In *Proceedings of the International Workshop on Public Data about Software Development*, 2007.

[15] N. Ducheneaut. Socialization in an open source software community: A socio-technical analysis. In *Proceedings of the European Conference on Computer Supported Cooperative Work*, 2005.

[16] M. Fischer, M. Pinzger, and H. Gall. Populating a release history database from version control and bug tracking systems. In *Proceedings of the International Conference on Software Maintenance*, 2003.

[17] M. Godfrey and Q. Tu. Evolution in open source software: A case study. In *Proceedings of the International Conference on Software Maintenance*, 2000.

[18] A. E. Hassan. Predicting faults using the complexity of code changes. In *Proceedings of the International Conference on Software Engineering*, 2009.

[19] L. Hattori and M. Lanza. On the nature of commits. In *Proceedings of the International Conference on Automated Software Engineering-Workshops*, 2008.

[20] A. Hindle, D. German, M. Godfrey, and R. Holt. Automatic classication of large changes into maintenance categories. In *Proceedings of the International Conference on Program Comprehension*, 2009.

[21] A. Hindle, D. German, and R. Holt. What do large commits tell us?: A taxonomical study of large commits. In *Proceedings of the International Working Conference on Mining Software Repositories*, 2008.

[22] J. Howison and K. Crowston. The perils and pitfalls of mining SourceForge. In *Proceedings of the International Workshop on Mining Software Repositories*, 2004.

[23] W. Humphrey. *Managing the software process*. Addison-Wesley Reading, MA, 1989.

[24] S. Koch. Evolution of open source software systems–a large-scale investigation. In *Proceedings of the International Conference on Open Source Systems*, 2005.

[25] J. L. Krein, A. C. MacLean, D. P. Delorey, C. D. Knutson, and D. L. Eggett. Language entropy: A metric for characterization of author programming language distribution. In *Proceedings of the International Workshop on Public Data about Software Development*, 2009.

[26] J. L. Krein, A. C. MacLean, D. P. Delorey, C. D. Knutson, and D. L. Eggett. Impact of programming language fragmentation on developer productivity: A SourceForge empirical study. *International Journal of Open Source Software and Processes*, 2(2):41–61, 2010.

[27] A. C. MacLean, L. J. Pratt, J. L. Krein, and C. D. Knutson. Threats to validity in analysis of language fragmentation on SourceForge data. In *Proceedings of the International Workshop on Replication in Empirical Software Engineering Research*, 2010.

[28] A. C. MacLean, L. J. Pratt, J. L. Krein, and C. D. Knutson. Trends that affect temporal analysis using SourceForge data. In *Proceedings of the International Workshop on Public Data about Software Development*, 2010.

[29] A. K. McCallum. MALLET: A machine learning for language toolkit. Accessed, Feb 2013. <`http://mallet.cs.umass.edu`>.

[30] P. McDonald, D. Strickland, and C. Wildman. Estimating the effective size of autogenerated code in a large software project. In *Proceedings of the International Forum on COCOMO and Software Cost Modeling*, 2002.

[31] A. Mockus, R. T. Fielding, and J. D. Herbsleb. Two case studies of open source software development: Apache and Mozilla. *Transactions on Software Engineering and Methodology*, 11(3):309–346, 2002.

[32] K. Nakakoji, Y. Yamamoto, Y. Nishinaka, K. Kishida, and Y. Ye. Evolution patterns of open-source software systems and communities. In *Proceedings of the International Workshop on Principles of Software Evolution*, 2002.

[33] L. J. Pratt, A. C. MacLean, C. D. Knutson, and E. K. Ringger. Cliff walls: An analysis of monolithic commits using latent dirichlet allocation. In *Proceedings of the International Conference on Open Source Systems*, 2011.

[34] A. Rainer and S. Gale. Evaluating the quality and quantity of data on open source software projects. In *Proceedings of the International Conference on Open Source Systems*, 2005.

[35] E. S. Raymond. The cathedral and the bazaar. *Knowledge, Technology & Policy*, 12(3):23–49, 1999.

[36] A. Tarvo. Mining software history to improve software maintenance quality: A case study. *IEEE Software*, 26(1):34–40, 2009.

[37] S. Uchida, A. Monden, N. Ohsugi, T. Kamiya, K. Matsumoto, and H. Kudo. Software analysis by code clones in open source software. *The Journal of Computer Information Systems*, 2005.

[38] M. Van Antwerp and G. Madey. Advances in the SourceForge research data archive (srda). In *Proceedings of the International Conference on Open Source Systems*, 2008.

[39] H. Wallach, D. Mimno, and A. McCallum. Rethinking lda: Why priors matter. In *International Conference on Neural Information Processing Systems*, 2009.

[40] J. Xu, Y. Gao, S. Christley, and G. Madey. A topological analysis of the open souce software development community. In *Proceedings of the Annual Hawaii International Conference on System Sciences*, 2005.

[41] T. Zimmermann and P. Weißgerber. Preprocessing cvs data for fine-grained analysis. In *Proceedings of the International Workshop on Mining Software Repositories*, 2004.

[42] T. Zimmermann, A. Zeller, P. Weissgerber, and S. Diehl. Mining version histories to guide software changes. *IEEE Transactions on Software Engineering*, 31(6):429–445, 2005.