Trends That Affect Temporal Analysis Using SourceForge Data

Alexander C. MacLean, Landon J. Pratt, Jonathan L. Krein, Charles D. Knutson
Brigham Young University
2204 TMCB
Provo, UT 84602
amaclean@byu.edu, hndsoff8@cs.byu.edu, jonathankrein@byu.net,
knutson@cs.byu.edu

ABSTRACT

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.

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. [15, 4]

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 [3]. 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., [1, 5, 9, 11, 14, 17, 20]). 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 [16], recent studies suggest that the two may not be as different as originally thought [2, 7]. Further, as one of the largest OSS hubs, SourceForge hosts thousands of projects—providing extensive data on thousands of mature projects [6]. 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

[1]The number of subscribers to the SRDA (SourceForge Research Data Archive) currently exceeds 100 [19]. The actual number of researchers engaging SourceForge data is likely several times that.

originated from efforts to replicate the results of a previous study [11, 12]. 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.*

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 [5].

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.

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 1). In our analysis we group commits into days to identify cliff walls programmatically.

2.2.1 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 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.

[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%.

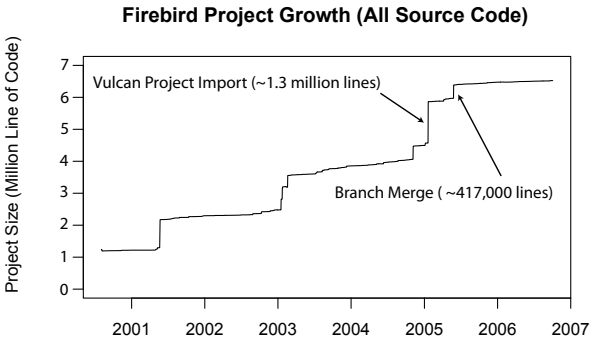


Figure 1: Growth of Firebird over time.

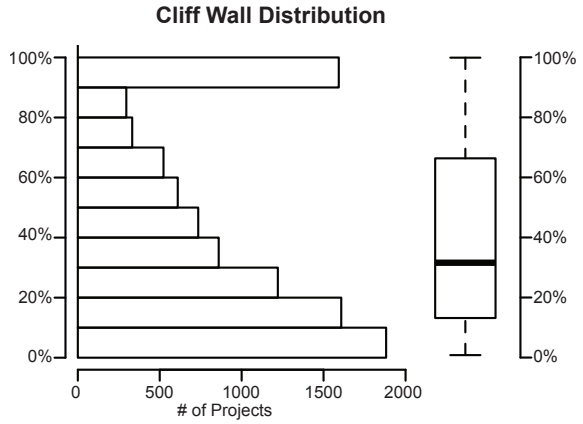


Figure 2: Distribution of projects by largest cliff walls. One outlier has been removed.[2]

Cliff walls appear in all phases of project growth. In Figure 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). After the initial source commit (2 1/2 years after the project was created) JXPFW appears to grow normally.

2.2.2 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 we discuss four of those reasons.

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

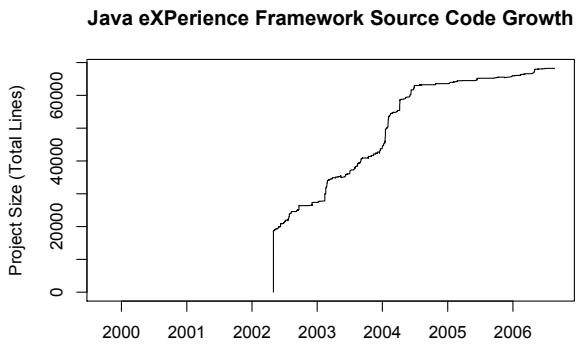


Figure 3: Growth of the Java eXPerience Frame-Work over time.

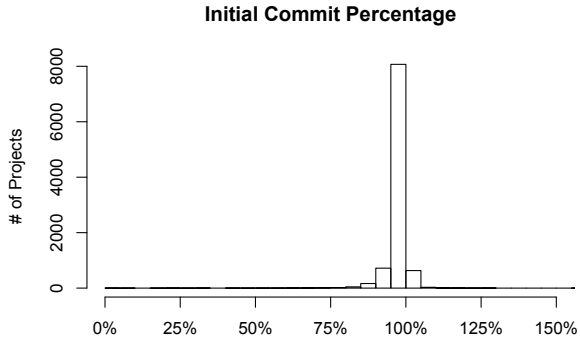


Figure 4: Distribution of projects by Initial Commit Percentage.

lines of code, is distinct from lines of code committed to (or deleted from) a preexisting file.

2.3.1 *Anomaly Description*

Initial Commit Percentage (ICP) is the percentage of the total size of the project that is made up of initial commits. Figure 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 5; note the log scale on the y-axis). However, this distribution holds, with small variation, regardless of project size (see Figure 6). High ICP indicates that revisionary changes to existing files constitute a small percentage of project growth.

2.3.2 *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. REASONS FOR PROBLEMS

Although there are many possible causes for the anomalies mentioned in Section 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

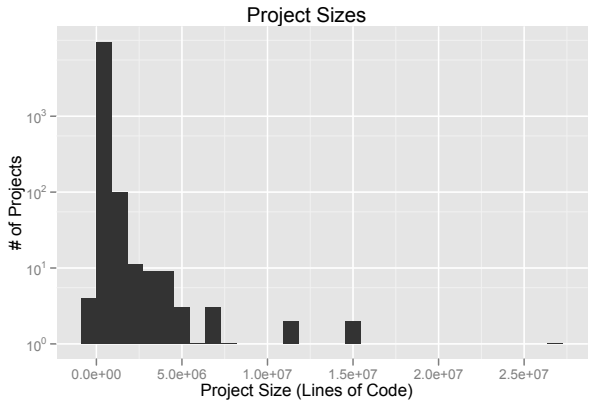


Figure 5: Project sizes.

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.

3.1 Off-line (Internal) Development

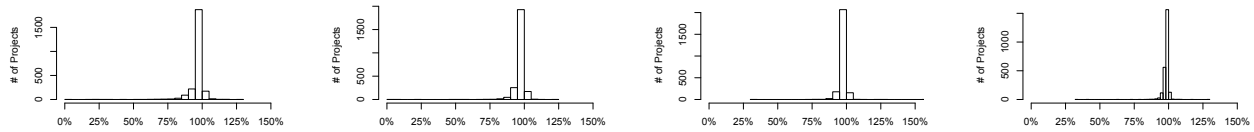
Many projects in our data set are committed as finished, monolithic entities. After the initial commit the authors commit infrequently and in large chunks. They do not commit frequent, incremental changes that capture development at a fine granularity. In essence, these projects use SourceForge as a 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" [10].

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 3.4) within the SourceForge CVS repository which the gatekeeper inspects before merging it into

[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".



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

Figure 6: Distribution of project by Initial Commit Percentage discretized by project size quartile.

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.

3.2 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 [13]). 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 [13]. 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 [18]. 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 Project Imports

In Figure 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.

3.4 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 [21].

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 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.

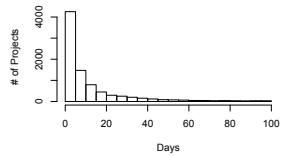
4. SOLUTIONS

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.

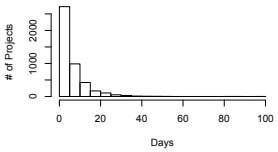
4.1 Identify Merges

In Section 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 ex-



(a) All Projects



(b) > 40 commits

Figure 7: Distribution of projects by frequency of author commits.

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

Table 1: Project Size Quartiles (Lines of Code)

amine each transaction for which the log message contains the word "merge" and determine if the transaction is indeed the merge of a branch [21]. 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 [8]. This approach is computationally intensive and may not scale to studies of large sets of projects.

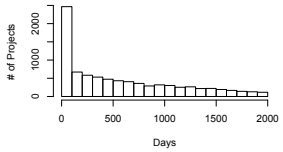
We suggest the possibility of a third method, that of simply assuming that all revisions containing the "merge" keyword are merges. This is the fastest method that we have yet identified, but would also likely suffer in terms of accuracy. Future work is required to establish the best method(s) for identifying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

4.2 Author Behavior

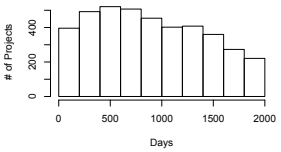
One way to identify project records that contain fine grained evolutionary data is to filter for projects that have authors who "commit early, commit often." *Frequency of commits* is a metric that captures this behavior. Figure 7 illustrates the distribution of projects by commit frequency. We also show the distribution for projects with more than 40 commits 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 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.

4.3 Project Size

Small projects have a much higher occurrence of large cliff walls than large projects. Figure 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



(a) All Projects



(b) > 40 commits

Figure 8: Distribution of projects by project life span: the time between the first and the last commit in a project.

commit. Interestingly, all of the histograms in Figure 9 have a spike at 100%. However, 9(b), 9(c), and 9(d) have successively greater area under the curve towards 0% (see Table 1 for quartiles). This 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.

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 7 and 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.

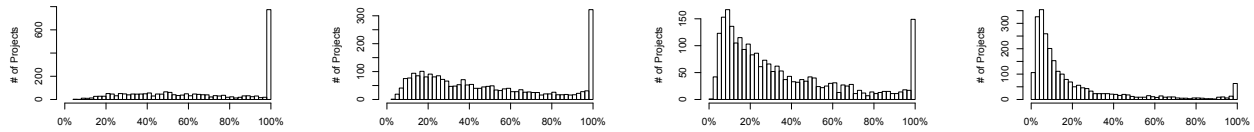
6. REFERENCES

[1] Christian Bird, Alex Gourley, Prem Devanbu, Anand Swaminathan, and Greta Hsu. Open borders? immigration in open source projects. *Mining Software Repositories, International Workshop on*, 0:6, 2007.

[2] Christian Bird, David Pattison, Raissa D'Souza, Vladimir Filkov, and Premkumar Devanbu. Latent social structure in open source projects. In *16th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 24–35, New York, NY, USA, 2008. ACM.

[3] 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, Sep/Oct 1999.

[4] Cleidson de Souza, Jon Froehlich, and Paul Dourish. Seeking the source: Software source code as a social and technical artifact. In *GROUP '05: Proceedings of the 2005 International ACM SIGGROUP Conference on Supporting Group Work*, pages 197–206, New York, NY, USA, 2005. ACM.



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

Figure 9: Distribution of projects by largest cliff wall as a percentage of project size. See Section 2.2.1 for a discussion of how to read these histograms.

[5] Daniel P. Delorey, Charles D. Knutson, and Scott Chun. Do programming languages affect productivity? a case study using data from open source projects. In *1st International Workshop on Emerging Trends in FLOSS Research and Development (FLOSS '07)*, May 2007.

[6] Daniel P. Delorey, Charles D. Knutson, and Christophe Giraud-Carrier. Programming language trends in open source development: An evaluation using data from all production phase sourceforge projects. In *2nd International Workshop on Public Data about Software Development (WoPDaSD '07)*, June 2007.

[7] Nicolas Ducheneaut. Socialization in an open source software community: A socio-technical analysis. *Computer Supported Cooperative Work (CSCW)*, 14(4):323–368, 2005.

[8] Michael Fischer, Martin Pinzger, and Harald Gall. Populating a release history database from version control and bug tracking systems. In *ICSM '03: Proceedings of the International Conference on Software Maintenance*, page 23, Washington, DC, USA, 2003. IEEE Computer Society.

[9] Ahmed E. Hassan. Predicting faults using the complexity of code changes. In *Proceedings of the 31st International Conference on Software Engineering (ICSE '09)*, pages 78–88, New York, NY, USA, 2009. ACM.

[10] J. Howison and K. Crowston. The perils and pitfalls of mining SourceForge. In *Proceedings of the International Workshop on Mining Software Repositories (MSR 2004)*, pages 7–11. Citeseer, 2004.

[11] Jonathan L. Krein, Alexander C. MacLean, Daniel P. Delorey, Charles D. Knutson, and Dennis L. Eggett. Language entropy: A metric for characterization of author programming language distribution. *4th Workshop on Public Data about Software Development*, 2009.

[12] Jonathan L. Krein, Alexander C. MacLean, Daniel P. Delorey, Charles D. Knutson, and Dennis L. Eggett. Impact of programming language fragmentation on developer productivity: a sourceforge empirical study. In *International Journal of Open Source Software and Processes (IJOSSP)*, Publication Pending.

[13] P. McDonald, D. Strickland, and C. Wildman. Estimating the effective size of autogenerated code in a large software project. In *Proceedings of the 17th International Forum on COCOMO and Software Cost Modeling*, 2002.

[14] Audris Mockus, Roy T. Fielding, and James D. Herbsleb. Two case studies of open source software development: Apache and mozilla. *ACM Transactions on Software Engineering and Methodology*, 11(3):309–346, 2002.

[15] Kumiyo Nakakoji, Yasuhiro Yamamoto, Yoshiyuki Nishinaka, Kouichi Kishida, and Yunwen Ye. Evolution patterns of open-source software systems and communities. In *IWPSE '02: Proceedings of the International Workshop on Principles of Software Evolution*, pages 76–85, New York, NY, USA, 2002. ACM.

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

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

[18] 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*, 45(3):1–11, 2005.

[19] M. Van Antwerp and G. Madey. Advances in the sourceforge research data archive (srda). In *Fourth International Conference on Open Source Systems, IFIP 2.13 (WoPDaSD 2008)*, Milan, Italy, September 2008.

[20] Jin Xu, Yongqin Gao, Scott Christley, and Gregory Madey. A topological analysis of the open souce software development community. *HICSS '05: Proceedings of the 38th Annual Hawaii International Conference on System Sciences*, 7, 2005.

[21] 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.