2012-02-27

Commit Patterns and Threats to Validity in Analysis of Open Source Software Repositories

Alexander Curtis MacLean
Brigham Young University - Provo

Commit Patterns and Threats to Validity in Analysis of Open Source Software Repositories

Alexander C. MacLean

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
Kevin D. Seppi
Robert P. Burton

Department of Computer Science

Brigham Young University

April 2012

ABSTRACT

Commit Patterns and Threats to Validity in Analysis of Open Source Software Repositories

Alexander C. MacLean
Department of Computer Science, BYU
Master of Science

In the course of studying the effects of programming in multiple languages, we unearthed troubling trends in SourceForge artifacts. Our initial studies suggest that programming in multiple languages concurrently negatively affects developer productivity. While addressing our initial question of interest, we discovered a pattern of monolithic commits in the SourceForge community. Consequently, we also report on the effects that this pattern of commits can have when using SourceForge as a data-source for temporal analysis of open source projects or for studies of individual developers.

Keywords: Open source, artifact, software engineering, threats to validity

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 The Changing Landscape of Open Source Software Engineering Research

The late 1990's saw the emergence of "forges," large collections of open source software projects. Prominent open source software repositories established during this timeframe include the Apache Software Foundation (June 1999), SourceForge (November 1999), and the Eclipse Board of Stewards (November 2001). By the end of 2009, 26 major forges supported development communities whose members number in the millions [38]. The largest of these, SourceForge, hosts more than 270,000 open source projects and 3 million developers.

Historically, empirical software engineering researchers have studied relatively small numbers of projects. Analysis of even a dozen projects represented a vast[1] trove of information. The availability of thousands of project archives through open source software forges represents a massive archaeological record that researchers are free to inspect. This unprecedented access to project information has fundamentally changed the empirical software engineering research landscape.

1.2 Limitations of Traditional Software Engineering Research

To understand the context that precipitated the euphoric response to this new glut of data, we first explore limitations in traditional methods of software engineering research. Historically, the only option available to software engineering researchers was to manually gather data from companies through interviews, code inspections, and other intrusive procedures. Here

[1] . . . ye scurvy dogs, we seek a treasure. . .

we address three specific limitations to these methods: 1) small sample size, 2) research overhead, and 3) sampling bias. In each of the sections that follow, we discuss the ways in which these limitations are mitigated or eliminated through the use of artifact data.

1.2.1 Small Sample Size

The most valuable assets within a software company are the engineers. In the process of building software, engineers develop mental models that are subsequently translated into code. These mental models are difficult to transfer between engineers and are typically obtained by an engineer through working with the codebase[2]. The formation and maintenance of mental models renders individual engineers indispensable to projects, particularly during key phases. Needless to say, indispensable engineers are not generally available to be studied.

Artifact-based research using online forges removes this barrier. In essence, data is gathered while the developers contribute to the project; each interaction with the source code management tool, online forums, or other online tools, is recorded as part of the normal work flow. Thus researchers may glean a wealth of information without negatively impacting project productivity.

1.2.2 Research Overhead

To provide data for a study, a developer must spend time with researchers that would otherwise be spent working on the project. In small organizations this dampening of productivity has a large effect on the overall output of the team. Not surprisingly, most traditional software engineering studies involving professional developers studied large organizations—generally government funded. A large organization with hundreds of developers can absorb the overhead imposed by researchers much more readily than can a small company.

[2]For an engineer new to a project, it takes a significant amount of time develop a mental model from existing code. The resultant derived mental model is twice removed from the original: 1) the code is an imperfect implementation of the original model, and 2) the mental model developed from the code is an imperfect interpretation of the functionality of the product.

As with small sample size, this limitation is non-existent when using forge data. All data is aggregated as a natural byproduct of software development so research studies impose no overhead on the development process.

1.2.3 Sampling Bias

Statistical extrapolation requires a sample that has been randomly selected from a target population. However, most companies are reluctant to sacrifice employee productivity on the altar of research without some clear (and ideally, short term) return on investment. Consequently, in traditional empirical software engineering studies, researchers select a sample of professional software engineers from a group of companies to which they have access. Results drawn from this sample can only be extrapolated to those companies from which the sample was drawn.

To avoid these difficulties, some studies use college students drawn from one or more university classes as subjects [7, 13, 43, 49]. This selection strategy allows one to extrapolate the results of such studies only to the classes from which the students were drawn and not to the population of professional software developers.

In short, the population of developers generally available for study (college students) is not representative of the target population that we actually desire to understand (professional software developers). Traditional empirical software engineering research is plagued by this unavoidable tradeoff.

Utilizing open source data removes this sampling dilemma; all work contributed by developers is available to researchers.

1.3 Forges

Researchers quickly realized that online forges contain a wealth of data that indirectly paints a picture of development practices while mitigating the aforementioned difficulties.

1.3.1 Research Community

In 2004, five years after the creation of the Apache Software Foundation, and three years after the formation of the Eclipse Board of Stewards, the International Conference on Software Engineering (ICSE) sponsored the first Working Conference on Mining Software Repositories (MSR) in Edinburgh, Scotland. The stated purpose of the conference was "to consider methods to use data stored in software repositories to further understanding of software development practices" [2]. MSR continues to colocate with ICSE each year.

In 2005, the first International Conference on Open Source Systems (OSS) was held in Genova, Italy, as an ongoing "forum to discuss theories, practices, experiences, and tools on development and applications of OSS systems" [3]. OSS has been held each year since.

In 2006, OSS sponsored the first Workshop on Public Data about Software Development (WoPDaSD) in Como, Italy, "to foster the production and analysis of publicly available data sources about software development and the exchange of data between different research groups" [4]. WoPDaSD was held in conjunction with OSS for five years before its topics were subsumed by the conference, obviating the need for a separate workshop.

In 2007, the first International Workshop on Emerging Trends in FLOSS Research and Development (FLOSS) was held in Irvine, California as a National Science Foundation sponsored invitation-only event. The second FLOSS workshop was held in Irvine, California in 2010.

The growing OSS research community is composed of Computer Scientists, Economists, and Social Scientists. Although members of each discipline approach the questions in Open Source Systems differently, they share two common goals: to understand the developers who contribute to open source projects and the organizations within which they operate.

1.3.2 Enter SEQuOIA

In 2006, Dr. Charles Knutson established the BYU SEQuOIA[3] lab to empirically study software development. Employing a method they referred to as "Software Archaeology," lab researchers began sifting through open source forges in search of insights into the software development process.

In 2007, Dan Delorey published the first paper of the newly founded lab, using data from 9,999 projects (every "Production/Stable" or "Maintenance" phase project hosted on SourceForge at that time) [19]. Over the next four years, members of the lab published seven additional papers based upon analysis of the same data set, including the four papers presented in this thesis.

1.4 Thesis

Thesis Statement: Using publicly available artifacts from online source code repositories, we can quantify the impact on developer productivity of writing software in multiple programming languages.

Online source code repositories contain meta data concerning file changes, including the author of the change, the date of the change, and the amount of source code modified. Using this meta data, we can reconstruct a picture of developer contribution patterns over time.

We present four papers that address our thesis statement. The first two papers present two studies in which we used data from SourceForge to identify a negative correlation between developer productivity and coding in multiple programming languages. These findings support the thesis statement. The following two papers explore challenges that we identified while using SourceForge data.

[3]Software Engineering Quality: Observation, Insight, and Analysis

1.4.1 Language Entropy: A Metric for Characterization of Author Programming Language Distribution

In 2009, we developed a metric, which we called *Language Entropy*, to measure the degree to which a single developer utilizes multiple languages. High language entropy indicates that a developer works evenly in multiple languages; low language entropy indicates that the developer works primarily in a single language. A key feature of this metric is its dramatic response to the introduction of additional languages. The resulting paper, "Language Entropy: A Metric for Characterization of Author Programming Language Distribution," introduced the metric and presented a preliminary study. Our results suggest a negative correlation between high developer language entropy (a developer utilizing multiple languages evenly) and developer productivity (measured in lines of code). Chapter 2 is the full version of the paper which was presented at the Fourth International Workshop on Public Data about Software Development in Skövde, Sweden, on 6 June, 2009 (WoPDaSD 2009) [32].

1.4.2 Impact of Programming Language Fragmentation on Developer Productivity: a SourceForge Empirical Study

After validating Language Entropy as a metric, we visited a new subset of the SourceForge data to examine our question of interest: does working in multiple programming languages (and paradigms) impose a cognitive burden on the developer? The resulting paper, "Impact of Programming Language Fragmentation on Developer Productivity: a SourceForge Empirical Study," confirmed the results of our pilot study, that switching between languages had a statistically significant dampening effect on the output of a developer. Chapter 3 is the full version of the paper which was published in the International Journal of Open Source Software and Processes (IJOSSP, June, 2010) [33].

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

In early 2010 we began a replication of the Language Fragmentation study, seeking further enlightenment by approaching the question with fresh data and from a different angle. Within days we were enlightened by a realization that our data was significantly biased, and that certain of our assumptions were in fact misguided. We had unwittingly assumed that projects on SourceForge would mimic the "Linux model" where all development occurs in the open [46]. We were very, very wrong. While certain open source projects exhibit fine-grained commit patterns that represent daily development efforts, many others appear to use SourceForge primarly as a distribution mechanism, rather than as an active development repository. Chapter 4 is the full version of the paper, "Threats to Validity in Analysis of Language Fragmentation on SourceForge Data," which was presented at the 1st International Workshop on Replication in Empirical Software Engineering Research in Cape Town, South Africa, on 4 May, 2010 (RESER 2010) [34].

1.4.4 Trends That Affect Temporal Analysis Using SourceForge Data

Having identified and examined threats in our data, we next sought to mitigate some of these threats while preserving the ability to use the data to answer our questions of interest. In the resulting paper, "Trends That Affect Temporal Analysis Using SourceForge Data," we sought to identify sources of these anomalous patterns and suggest methods researchers may employ to mitigate their effects on statistical analysis. Chapter 5 is the full version of the paper presented at the Fifth International Workshop on Public Data about Software Development at Notre Dame, on 2 June, 2010 (WoPDaSD 2010) [35].

1.5 Going Forward

The work contained in this thesis sheds light on an important aspect of software archaeology: the need to understand the relationship between the development process that produces software and the data reflected in an open source forge as a result of that development effort.

Analyzing developer behavior through the study of residual artifacts is a boon to software engineering researchers who require large sets of data. Utilizing online forges allows us to make grounded statements about the activities of individual developers and the online organizations within which they operate. However, these conclusions must be based upon a solid understanding of the forges and repositories from which the data is drawn. Further replications of these studies need to examine the nature and personality of other forges, explore the difference between "open source" and "open development," and develop a more refined taxonomy with which to frame our conclusions.

Chapter 2

Language Entropy: A Metric for Characterization of Author Programming Language Distribution

Programmers are often required to develop in multiple languages. In an effort to study the effects of programming language fragmentation on productivity—and ultimately on a programmer's problem solving abilities—we propose a metric, *language entropy*, for characterizing the distribution of an individual's development efforts across multiple programming languages. To evaluate this metric, we present an observational study examining all project contributions (through August 2006) of a random sample of 500 SourceForge developers. Using a random coefficients model, we found a statistically significant correlation (alpha level of 0.05) between language entropy and the size of monthly project contributions (measured in lines of code added). Our results indicate that language entropy is a good candidate for characterizing author programming language distribution.

2.1 Question of Interest

What effect does working in multiple programming languages concurrently have on a programmer's productivity?

- Positive Correlation: A programmer contributing in multiple programming languages may be more productive due to his or her ability to draw from multiple programming paradigms. For example, software developers writing in a functional language such as Lisp arguably approach a problem differently than those writing in a purely object-oriented language such as Java.

- Negative Correlation: A developer contributing in more than one language may be less productive because he or she has to context switch between multiple languages.
- No Correlation: A developer's productivity may be independent of his or her programming language distribution.

2.2 Language Entropy

In order to empirically evaluate the correlation between language fragmentation and programmer productivity, we require a metric that accurately characterizes the distribution of an author's development efforts across multiple programming languages. In this section we present language entropy as a candidate metric, detail its calculation, and explain its behavior in response to changes in the number of languages a developer uses. For a deeper treatment of entropy as it relates to software engineering, see [51].

2.2.1 Definition

Entropy is a measure of chaos in a system. The concept of entropy originated in thermodynamics but has been adopted by information theory [27]. For our purposes, we use entropy as a measure of the evenness with which an author contributes in different programming languages. For example, if an author is working in two languages and splits his or her contribution evenly between the two, entropy is 1. However, a 75-25 split across the two languages yields an entropy of approximately 0.8 (see Table 2.1).

% Contribution		
A	**B**	**Entropy**
0	100	0
25	75	~ 0.8
50	50	1
75	25	~ 0.8
100	0	0

Table 2.1: Entropy Example for two languages, A and B.

2.2.2 Calculation

The general form for entropy is shown in Equation 3.1.

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

In this equation, the variables are defined as follows:

- S: the system
- c: the language count
- p: the proportion of the contribution of language i to the total contributions S

The general form of entropy can be applied to any number of languages to generate an entropy value. Two languages, as shown in Figure 3.1, produce a parabolic curve. Three languages produce a three-dimensional shape. Entropy calculations beyond three dimensions are difficult to visualize.

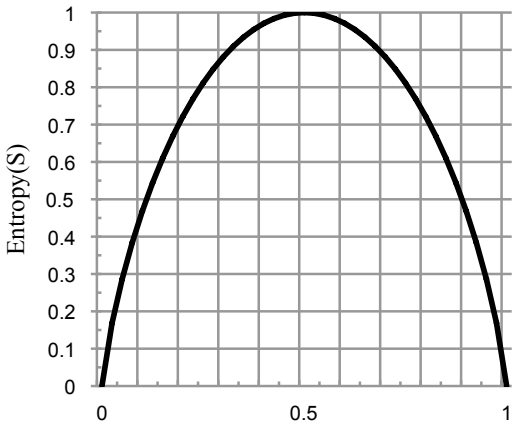


Figure 2.1: 2nd Order Entropy Curve

To compute an author's language entropy we calculate the proportion for each programming language represented in the author's total contribution—p_i values in Equation 3.1[1]. We then calculate the result of Equation 3.1 using those language proportions.

[1]For the purposes of this paper, contribution is defined as the number of lines of code produced per month.

2.2.3 Behavior

Language entropy characterizes the developer's fragmentation across multiple programming languages. However, because entropy is based on logarithms, its response to changes in the number of languages a developer uses is non-linear. The equation for the maximum possible entropy value for a given number of languages is shown in Equation 3.2, where c is the language count.

$$E_{max} = log_2(c) \tag{2.2}$$

Notice that the equation's maximum value increases as c increases. Thus, for each additional language in the entropy calculation, an author's maximum possible entropy value rises.[2] However, the effect of adding an additional language diminishes as the total number of languages increases (see Equation 3.3 and Table 3.1).

$$\lim_{c \to \infty} E_{max}(c+1) - E_{max}(c) = 0 \tag{2.3}$$

Conversely, the minimum possible language entropy is always 0, indicating that the author only added lines of code in a single language (see Table 3.1).

# of Languages	Min. Entropy	Max. Entropy
1	0	0
2	0	1
3	0	1.585
4	0	2
5	0	2.322
.	.	.
.	.	.
.	.	.
49	0	5.615
50	0	5.644

Table 2.2: Sample Entropy Values

[2]Note that the log operation is undefined at zero; thus, languages with a $p_i = 0$ must be excluded from the calculation.

2.3 Data

The data set used in this study was previously collected for a separate, but related work. It was originally extracted from the SourceForge Research Archive (SFRA), August 2006. For a detailed discussion of the data source, collection tools and processes, and summary statistics, see [21].

2.3.1 Description of the Data Set

The data set is composed of *all* SourceForge projects that match the following four criteria: 1) the project is open source; 2) the project utilized CVS for revision control; 3) the project was under active development as of August 2006; 4) the project was in a Production/Stable or Maintenance stage. The data set includes nearly 10,000 projects with contributions from more than 23,000 authors who collectively made in excess of 26,000,000 revisions to roughly 7,250,000 files [21].

A study by Delorey, Knutson, and Chun [19] identified more than 19,000 different file extensions in the data set, representing 107 unique programming languages. The study also noted that 10 of those 107 languages are used in 89% of the projects, by 92% of the authors, and account for 98% of the files, 98% of the revisions, and 99% of the lines of code in the data set. Table 3.2 shows the 10 most popular languages with rankings. Delorey et al. ranked the languages based on the following 5 factors: 1) total number of projects using the language; 2) total number of authors writing in the language; 3) total number of files written in the language; 4) total number of revisions to files written in the language; and 5) total number of lines written in the language.

	Project Rank	Author Rank	File Rank	Revision Rank	LOC Rank	Final Rank
C	1	1	2	2	1	1
Java	2	2	1	1	2	2
C++	4	3	4	4	3	3
PHP	5	4	3	3	4	4
Python	7	7	5	5	5	5
Perl	3	5	9	9	6	6
JavaScript	6	6	6	8	10	7
C#	9	9	7	6	7	8
Pascal	8	10	8	7	8	9
Tcl	11	8	10	10	9	10

Table 2.3: Top ten programming languages by popularity rankings

2.3.2 Producing a Data Sample

From the initial data set we extracted a random sample of 500 developers[3] along with descriptive details of all revisions that those developers made since the inception of the projects on which they worked. We then condensed this sample by totaling the lines of code added by each developer for each month in which that developer made at least one code submission. The final step in generating the sample was calculating the language entropy in each month for each developer. Note that months in which developers made no contributions are discarded due to the fact that the language entropy metric is undefined for zero lines of code.

Ignoring a developer's "inactive" months is reasonable since for this study we are more interested in whether lines of code production is related to language entropy than we are in the actual magnitude of that relationship. However, our model does assume that the code was written in the month in which it was committed. Therefore, months without submissions represent a confounding factor in this study.

To help account for multi-month code submissions, as well as the factors identified in [19], we applied several filters to the data sample. However, analyses of the filtered and unfiltered data produced approximately equivalent results. Therefore, we report our results from the more robust, unfiltered data sample.

[3]For the purposes of this study, a *developer* is an individual who contributed at least one line of code in at least one revision.

To filter the data, we 1) removed all data points of developers who submitted more than 5,000 lines of code during at least three separate months, and 2) removed all remaining data points for which the month's submission was greater than 5,000 lines of code. The first filter was intended to remove project gatekeepers, who submitted code on behalf of other developers. If a developer was suspected of being a gatekeeper, all of his/her contributions were excluded. The second filter was designed to remove significant quantities of auto-generated code.

We feel that these two filters are sufficient on the grounds that in [19], Delorey et al. ultimately controlled for outliers by capping the annual author contribution at 80,000 lines of code. Our limit of 5,000 lines of code per month results in a maximum possible annual contribution of 60,000 lines of code per author—a bit more conservative.

2.4 Analysis

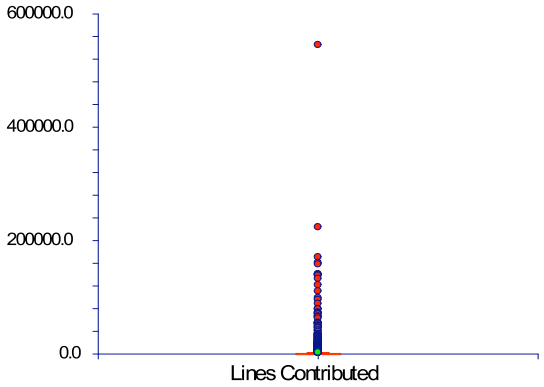


Figure 2.2: Box Plot of Lines Added

2.4.1 Transforming the Data

Figure 3.2(a) shows a box plot of the lines added. Three threats to statistical model assumptions are clearly visible: significant outliers, a skewed distribution, and a large data range. We adjust for all three issues by applying a natural log transformation. Notice in

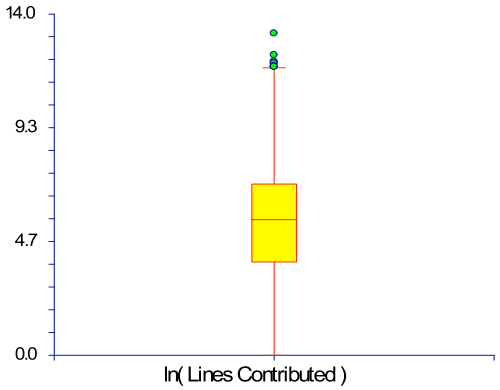


Figure 2.3: Box Plot of ln(Lines Added)

figure 3.2(b), which depicts the transformed data, that there are only minimal outliers, the range is controlled, and the distribution is approximately normal.[4]

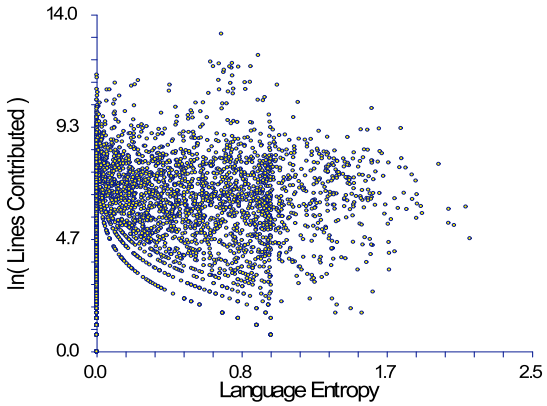


Figure 2.4: Plot of ln(Lines Added) vs. Language Entropy

2.4.2 Selecting a Statistical Model

Figure 3.3 displays a plot of lines added (on the natural log scale) versus language entropy, in which each point on the graph represents one month of work for one developer. First, be aware that the volume and distribution of data points (see table 3.3) is masked by crowding, which causes points to be plotted over other points. In total, there are 3,940 points plotted,

[4]Statistical models assume specific characteristics about data. Sometimes data must be transformed before it can be accurately analyzed. However, interpretation of the results must reflect the transformation. In this study, for instance, the slope of a regression line must be interpreted as a multiplicative factor since the dependent variable is logged.

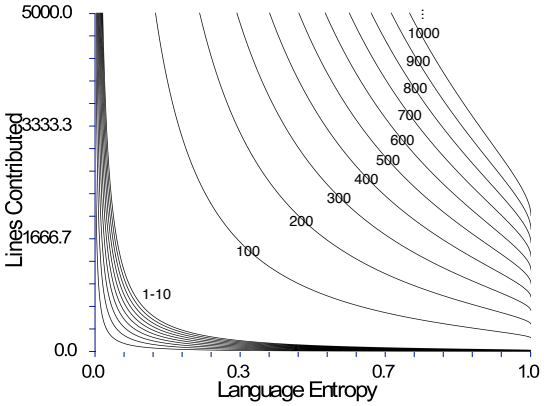


Figure 2.5: Graph of the First 10 Entropy Curves for the 2 Language Case

LE Range	**Data Points**
$LE = 0$	1,945
$0 < LE \leq 1$	1,705
$1 < LE$	290
Total Data Points:	3,940

Table 2.4: Distribution of Data Points by Language Entropy (LE)

of which 1,945 points lie on the y-axis at the entropy value of zero. Thus, nearly half the data consists of months in which developers submitted code in only one language.[5]

Further, there is a pattern of curving lines visible at the bottom of the point cloud between zero and one entropy. The banding pattern is due to both the nature of the language entropy calculation and the lines added. Specifically, the two metrics partition the data points into equivalence classes, one for each band on the graph. Data points in the first equivalence class—the band closest to the x-axes—correspond to monthly contributions in which all lines but one were written in the same language. Data points in the second equivalence class correspond to monthly contributions in which all but two lines were written in one language. By the fourth equivalence class the bands are so close that they blend together on the graph. Figure 3.5 shows a graph of the first 10 equivalence classes.

[5]The distribution of the data points with respect to language entropy is fairly consistent with [20], in which the authors (referring to the data set of this study) note that for approximately 65% of projects developers submit code in a single language per year. For 20% and 12% of projects, developers submit code in two and three languages per year respectively.

Parameter	Estimate	Lower 95% CL	Upper 95% CL	p-value	Standard Error	DF
zeroEntropyGroupMean	4.0690	3.9208	4.2172	¡.0001	0.07542	425
nonZeroEntropyGroup	2.2870	2.1014	2.4726	¡.0001	0.09411	196
nonZeroEntropySlope	-0.6963	-0.9646	-0.4280	¡.0001	0.13600	181

Table 2.5: Model Parameter Estimates

The scatter plot also exhibits a vertical boundary of points just before entropy of one. This pattern is possibly due to the sparseness of the data beyond entropy of one (only 290 points).

It is immediately apparent that the distribution of the data is different during months in which developers contributed code in only one language (zero entropy), versus months in which they contributed code in more than one language (greater than zero entropy). Therefore, it would be inappropriate to apply a simple linear regression model to the full range of the data. Instead, we use a *random coefficients model* which allows us to estimate a mean for the group at zero entropy, as well as fit a regression line to the rest of the data. These two groups could be analyzed separately, but fitting them under one model allows us to pool the data when computing the error terms, which results in tighter confidence intervals and a more efficient analysis.

2.4.3 Adjusting for Serial Correlation

A final concern is the potential for serial correlation in the data (i.e., the data correlates with itself) as a result of the measurements being taken over time. Estimating the mean of data that is self-correlated requires statistical adjustment in order to produce accurate results. The data sample in this study contains an average of eight months of measurements per developer, which is insufficient to confidently identify a serial correlation. However, to be conservative we assume serial correlation is present in the data and account for it in our analysis.

2.5 Results

Table 3.4 shows estimates (on the natural log scale) of the model parameters, with confidence intervals and two-sided p-values. All three parameters are statistically significant with p-values less than 0.0001. Such small p-values allow us to confidently conclude that the relationship between language entropy and lines added is *not* due to random chance. The low error terms, which result in narrow confidence intervals around the parameter estimates, give us confidence that our sample size is sufficient to accurately estimate the population variance. Further, since the data sample was randomly selected (as described in section 3.5.1), we conclude that the patterns in the data sample characterize the entire SourceForge population. However, since this is an observational study, we cannot infer causality. Therefore, the remainder of the discussion of results describes the observed relationship between language entropy and lines added.

In Table 3.4, the *zeroEntropyGroupMean* is an estimate of the mean of the data points at zero language entropy (the zero group, or ZG). The *nonZeroEntropyGroup* represents the estimated difference between the ZG mean and the intercept of the regression line for the non-zero entropy data (the non-zero group, or NZG). The very low p-value for this parameter indicates that the ZG mean is significantly different from the trend in the NZG. Adding the first two parameter estimates gives the estimate for the intercept of the NZG regression line (6.3560). The third parameter, *nonZeroEntropySlope*, represents the slope of the NZG regression line, which is negatively correlated with language entropy.

The magnitudes of these parameter estimates make more sense on the original scale. However, because the analysis is performed on log-transformed data, the back-transformed estimates must be interpreted differently. Specifically, the ZG mean and the intercept of the NZG regression line both represent medians on the original scale. Also, the slope of the NZG regression line becomes a multiplicative factor, which means that an increase in language entropy results in a multiplicative increase in lines added.

Thus, for months in which a developer submits code in one language (ZG), the developer contributes, on average, 58 lines of code (95% confidence interval from 50 to 68 lines of code). However, extrapolating the trend in the NZG, which represents months in which developers submitted code in more than one language, one would expect the ZG median to be 576 lines of code—a significant difference. Note, though, that this difference considers both highly and marginally active developers equally. The marginally active developers, who make only a few small contributions, and for whom a productivity increase is less interesting, may be significantly pulling down the ZG median (See section 3.9.4 for further discussion).

Lastly, for months in which a developer submits code in more than one language, the developer's monthly contributions decrease by an estimated 6.7% for each 0.1 unit increase in language entropy. For a 1.0 unit increase in language entropy, a developer's monthly contribution drops by approximately 50% on average.

2.6 Limitations

In the following subsections we identify several limitations of this study.

2.6.1 Non-Contributing Months

The developers in our data set did not always contribute to projects in contiguous months. For example, a developer might contribute changes in April, skip May, and contribute again in June. For the purposes of this study we assumed that developers submitted contributions in the same months in which those contributions were written. We took steps to help ensure our assumption (see Section 3.5.2). However, we do not have an empirical foundation for applying a cap of 5,000 lines to monthly programmer contributions. Also, we have not empirically validated our method of identifying gatekeepers.

2.6.2 SourceForge

Our inferences are limited to developers on SourceForge. Therefore, we cannot make general conclusions about other software development environments. Also, the SourceForge archive obscures certain information about developers (such as the identity of gatekeepers).

2.6.3 Productivity Measure

Despite its utility in this preliminary study, lines of code is a weak measure of programmer productivity. Further studies should extend the analysis of language entropy to other productivity models.

2.6.4 Marginally Active Developers

Developers who make only small contributions per month may bias the analysis results. Such developers are probably less likely to write in multiple languages in a given month, in which case filtering marginally active developers could reduce the disparity between the estimated mean of the group who wrote in only one language and the trend of the remaining data. Thus, it would be interesting to add an indicator variable to the model to distinguish such developers from those who regularly contribute more significant volumes of code.

2.7 Future Work

In this section we outline avenues for future research.

2.7.1 Establishing Causality

This study establishes a correlation between language entropy and the size of developer contributions for the SourceForge population. To understand the cause of the observed relationship we need to run controlled, randomized experiments. We believe that such efforts, in combination with corporate case studies (as described in section 3.10.2), will provide

meaningful results from which practitioners may make better-informed decisions regarding project-developer assignments and the adoption of new languages and frameworks.

2.7.2 Corporate Case Studies

Running a more robust analysis of language entropy utilizing data from industry projects would allow us to expand our inferences into the corporate domain, at which point we could ask a number of important questions, including:

- If my company is already maintaining a large code base in COBOL, how would my developers' productivity be affected by an additional project in Java?
- My company already supports products in different languages. Will my developers be more productive if I assign each one to a specific language, as opposed to spreading them across languages?

2.7.3 Paradigm Relationships

Many of the languages in our study cluster by paradigm (Java, C++, and C#, for example). Switching between programming languages that share a common paradigm may not be as cognitively difficult as switching between languages from different paradigms. We expect changes in entropy to affect a programmer working within a single paradigm less than one working across multiple paradigms.

2.7.4 Commonly Grouped Languages

In this study we examine the effect of language entropy on productivity across all languages. However, some languages are commonly used together (e.g., many web projects are based on Java, JavaScript, and HTML). Is the cognitive burden of context switching between languages reduced for developers who work across a set of commonly grouped languages?

2.7.5 Language Entropy as a Productivity Measure

To better understand the relationship between language entropy and other productivity metrics, we need to determine whether language entropy provides new information beyond the metrics already presented in the literature. If shown to be complementary, language entropy can be incorporated into more complex productivity models [14].

2.8 Conclusions

The results of this study suggest a correlation between language entropy and programmer productivity. However, because our study is observational, we cannot infer that the differences in language entropy caused the observed variation in productivity. Nevertheless, since the data was randomly selected, we can make inferences to the general SourceForge community for those developers who actively worked on Production/Stable or Maintenance projects from 1995 through August 2006. Specifically, we can make two inferences:

1. For those developers who wrote in multiple languages, higher language entropy is negatively correlated with the number of lines of code contributed per month.

2. For months in which developers submitted code in a single language, their contributions were significantly smaller than the trend suggested by the rest of the data.

The primary objective of this study was to develop a metric with which we could investigate the relationship between an author's ability to solve software problems and the distribution of programming languages within his or her project contributions. The relationship between language entropy and productivity in this initial study demonstrates that language entropy is a good candidate for measuring the distribution of an author's development efforts across multiple programming languages. This result, therefore, justifies further research into the relationship between language entropy and the problem-solving abilities of developers.

Chapter 3

Impact of Programming Language Fragmentation on Developer Productivity: A SourceForge Empirical Study

Programmers often develop software in multiple languages. In an effort to study the effects of programming language fragmentation on productivity—and ultimately on a developer's problem-solving abilities—we present a metric, *language entropy*, for characterizing the distribution of a developer's programming efforts across multiple programming languages. We then present an observational study examining the project contributions of a random sample of 500 SourceForge developers. Using a random coefficients model, we find a statistically (alpha level of 0.001) and practically significant correlation between language entropy and the size of monthly project contributions. Our results indicate that programming language fragmentation is negatively related to the total amount of code contributed by developers within SourceForge, an *open source software* (OSS) community.

3.1 Introduction

The ultimate deliverable for a software project is a source code artifact that enables computers to meet human needs. The process of software development, therefore, involves both problem solving and the communication of solutions to a computer in the form of software. We believe that the programming languages with which developers communicate solutions to computers may in fact play a role in the complex processes by which those developers generate their solutions.

Baldo et al. define language as a "rule-based, symbolic representation system" that "allows us to not simply represent concepts, but more importantly for problem solving, facilitates our ability to manipulate those concepts and generate novel solutions" [6]. Although their study focused on the relationship between *natural* language and problem solving, their concept of language is highly representative of languages used in programming activities. Other research in the area of linguistics examines the differences between mono-, bi-, and multilingual speakers. One particular study, focusing on the differences between mono- and bilingual children, found specific differences in the subjects' abilities to solve problems [8]. These linguistic studies prompt us to ask questions about the effect that working concurrently in multiple programming languages (a phenomenon we refer to as *language fragmentation*) has on the problem-solving abilities of developers.

In an effort to increase both the quality of software applications and the efficiency with which applications can be written, developers often incorporate multiple programming languages into software projects. Each language is selected to meet specific project needs, to which it is specialized—for instance, in a web application a developer might select SQL for database communication, PHP for server-side processing, JavaScript for client-side processing, and HTML/CSS for the user interface. Although language specialization arguably introduces benefits, the total impact of the resulting language fragmentation on developer performance is unclear. For instance, developers may solve problems more efficiently when they have multiple language paradigms at their disposal. However, the overhead of maintaining efficiency in more than one language may also outweigh those benefits. Further, development directors and programming team managers must make resource allocation, staff training, and technology acquisition decisions on a daily basis. Understanding the impact of language fragmentation on developer performance would enable software companies to make better-informed decisions regarding which programming languages to incorporate into a project, as well as regarding the division of developers and testers across those languages.

To begin understanding these issues, this paper explores the relationship between language fragmentation and developer productivity. In Sections 3.2 and 3.3 we define and justify the metrics used in the paper. We first discuss our selection of a productivity metric, after which we describe an entropy-based metric for characterizing the distribution of a developer's efforts across multiple programming languages. Having defined the key terms, Section 3.4 presents the thesis of the paper, and Sections 3.5 and 3.6 describe, justify, and validate the data and analysis techniques. We then present in Section 3.7 the results of an observational study of SourceForge, an *open source software* (OSS) community, in which we demonstrate a significant relationship between language fragmentation and productivity. Establishing this relationship is a necessary first step in understanding the impact that language fragmentation has on a developer's problem-solving abilities.

3.2 Productivity

According to the 1993 IEEE Standard for Software Productivity Metrics, "productivity is defined as the ratio of the output product to the input effort that produced it" [1]. Although this ratio may be as difficult to accurately quantify as problem-solving ability, it has been extensively studied in the context of Software Engineering.

In the 1960's, Edward Nelson performed one of the earliest studies to identify programmer productivity factors [41]. Nelson found that programmer productivity correlates with at least 15 factors. More recently (2000), Capers Jones identified approximately 250 factors that he claims influence programmer performance [31]. Summarizing this research, Endres and Rombach state that reducing productivity to "ten or 15 parameters is certainly a major simplification" [25]. With so many contributing factors to measure, it is not surprising that numerous productivity metrics have been proposed in the literature.

Nevertheless, all reasonable productivity metrics intercorrelate to some degree, and all productivity metrics suffer from threats to validity—the significance of those threats depends on the circumstances in which the metrics are applied. The researcher, therefore, must weigh

the trade-offs and select a suitable metric based on the available data and the context of the study. For a discussion of the trade-offs inherent in various common productivity metrics, as well as an overview of the primary threats to the validity of those metrics, we refer the reader to work by Conte, Dunsmore, and Shen [16] and to work by Endres and Rombach [25]. The most common software productivity metrics include function points and lines of code (LOC).

Function points attempt to measure software production by assigning quantitative values to software functionality. Points are accrued for each piece of functionality implemented in software, with more points assigned to more complex functionality. Function points are based on the idea that the ultimate goal of software is to meet specific human needs. Since human needs are formalized into project requirements, measuring the accomplishment of project requirements provides a good indication of progress. As such, function points are often applied to software requirements prior to coding in order to estimate needed resources. Thus, function points work well when measuring productivity for one or two projects, for which the requirements are well documented. Without requirements, as is the case in SourceForge data, calculating function points becomes much more difficult. Measuring functionality for thousands of projects is simply infeasible.

In the literature, the list of studies that rely on LOC and time primitives to estimate productivity is lengthy. Studies that use these primitives (e.g., [52] [12] [24] [15]) often justify the selection based on the availability and accessibility of data. Despite their popularity, LOC and time primitives are not without threats to validity.

The primary concerns with using LOC metrics to estimate productivity include:

1. LOC definitions differ by organization. For instance, are declarative statements counted, or executable statements only? Are physical lines counted or logical lines?
2. Coding styles vary by developer; some developers are more verbose.
3. When developers are aware that they are being measured, they may inflate their LOC scores.

4. The effort required to produce and incorporate new code is different from that of reused code.

5. Programming language verbosity varies based on syntax, built-in features, and the use of libraries (e.g., Perl regular expressions versus parsing C-strings).

The first of these threats is controlled for in this study by extracting all revision data from a common revision management system (CVS), which counts all lines added, modified, and deleted in a consistent manner across projects. We control for the second threat by examining trends within (rather than across) developers. Thus, we do not compare the data points of one developer directly against those of another (see Section 3.6.2). Concerning the third threat, we are confident that developers did not try to artificially inflate their LOC scores since the data was collected after the fact—developers had no prior knowledge of this study and little incentive to alter their normal habits. Further, OSS community norms would also tend to prevent developers from contributing large volumes of code, especially since such code would likely not be of high quality. To address the fourth threat, we applied filters to the data that help account for code reuse, but found no significant differences between the analyses of filtered and non-filtered data (see Section 3.5.2). *The last threat remains a limitation of this study.* To account for language verbosity we would need a method for normalizing the data, the development of which is beyond the scope of this paper (see Section 3.9.3).

When estimating input effort using time primitives, the primary concern is maintaining consistency across organizations. Which activities (e.g., requirements gathering, coding, maintenance), which people (e.g., direct and indirect project members), and which times (e.g., productive and unproductive) are counted? We control for time measurement variation as we did for the consistency issues of the LOC metric, by taking all data from CVS. Thus for all projects, we consistently count the coding and maintenance activities of direct project members during productive times.

Under these circumstances, and considering the availability of both LOC and time information in our SourceForge data, we use *developer code contribution per time-month* as a

productivity measure—where 1) *developer code contribution* is defined as the total number of lines modified within, or added to, all source code files, across all projects, by a particular developer (as reported by CVS), and 2) *time-month* refers to the literal months of the year, as opposed to measuring contribution per person-month. Strictly speaking, the time-month does not directly measure actual input effort, but due to data availability constraints we use it to approximate person-months. Recognizing this limitation, we believe that studying developers in aggregate helps control for monthly input effort variations.

Thus, although imperfect, code contribution per time-month is a reasonable productivity metric within the context of this study. Nevertheless, replicating this study using other productivity metrics may prove valuable.

3.3 Language Entropy

In order to empirically evaluate the correlation between language fragmentation and programmer productivity, we require a metric that effectively characterizes the distribution of a developer's efforts across multiple programming languages. In this section we present the *language entropy* metric developed by Krein, MacLean, Delorey, Knutson, and Eggett [32]. After defining the metric, we detail its calculation and explain its behavior in response to changes in the number and proportions of languages a developer uses. For a deeper treatment of entropy as it more broadly relates to software engineering, see work by Taylor, Stevenson, Delorey, and Knutson [51].

3.3.1 Definition

Entropy is a measure of disorder in a system. In thermodynamics, entropy is used to characterize the randomness of molecules in a system. Information theory redefines entropy in terms of probability theory [27] [47]. In this paper, we apply the latter interpretation of entropy to measure the evenness with which a developer's total contribution (to one or more software projects) is spread across one or more programming languages. Other works use

similar interpretations of entropy to measure various software characteristics [28] and [9], but none of them apply entropy to language fragmentation.

3.3.2 Calculation

The general formula for calculating the entropy of a system in information theory is shown in Equation 3.1, in which S is the system of elements, c is the number of mutually exclusive classes (or groupings) of the elements of S, and p_i is the proportion of the elements of S that belong to class i.

$$E(S) \;=\; -\sum_{i=1}^{c}(p_i \cdot log_2 p_i) \tag{3.1}$$

To apply this general entropy formula to language fragmentation, we specifically define the variables in Equation 3.1 as follows:

- S: a developer's total contribution (i.e., the number of lines modified within, or added to, all source code files by a particular developer)
- c: the number of programming languages represented in S
- p_i: the proportion of S represented by programming language i
- $E(S)$: the language entropy of the developer

For example, if a developer is working in two languages and splits his or her contribution evenly between the two, the entropy of the developer's total contribution is 1. However, a 75-25 split across the two languages yields an entropy value of approximately 0.8 (see Figure 3.1).

3.3.3 Behavior

Language entropy characterizes the distribution of a developer's efforts across multiple programming languages. Maximum entropy occurs when a developer produces code using programming languages in equal proportions. Thus, substituting $p_i = 1/c$ for all i in Equation 3.1, we arrive at a discrete function for the maximum possible entropy for a given number of

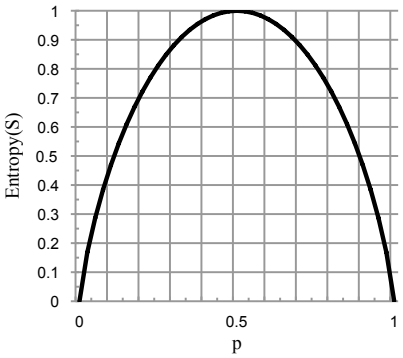


Figure 3.1: Relationship between entropy and language proportion for two languages

languages (see Equation 3.2).

$$E_{max} = log_2(c) \tag{3.2}$$

Notice in Equation 3.2 that E_{max} increases as c increases, such that for each additional language a developer uses, his or her maximum possible entropy value rises. However, because entropy is based on logarithms, its response to changes in the number of languages a developer uses is non-linear. Specifically, the effect on the entropy score of adding an additional language diminishes as the total number of languages increases (see Equation 3.3 and Table 3.1).

$$\lim_{c \to \infty} E_{max}(c+1) - E_{max}(c) = 0 \tag{3.3}$$

We believe this behavior is appropriate for studying programming language use because the impact of adding a new language to the working set of a developer who already programs in multiple languages is, in many respects, less than the impact on a developer who previously worked in only one language.

However, considering the case of a person who already knows multiple languages from the same paradigm (say imperative), it is unclear whether the addition of a new language from a different paradigm (say object-oriented) would, in reality, impact the developer less than the addition of the previous language from the familiar paradigm. More generally, we suspect that the addition of a language from an unfamiliar paradigm would result in a more

dramatic impact than would the addition of a language from a familiar paradigm (see Section 3.10.3).

Conversely, the minimum language entropy for all values of c is essentially zero[1], indicating that the developer contributed in only one language (see Table 3.1).

# of Languages	Min. Entropy	Max. Entropy
1	0	0.00
2	0	1.00
3	0	1.59
4	0	2.00
5	0	2.32
.	.	.
.	.	.
.	.	.
49	0	5.62
50	0	5.64

Table 3.1: Entropy ranges for sample language cases

The entropy metric is applicable to any number of languages. Two languages, as shown in Figure 3.1, produce a parabolic curve. Three languages produce a three-dimensional shape. Entropy calculations beyond three dimensions are difficult to visualize.

3.4 Objective

The primary objective of this study is to take a first step in establishing the effect that language fragmentation has on the problem-solving abilities of developers by addressing the question, "What is the relationship between programmer productivity and the concurrent use of multiple programming languages?"

Prior to this study we anticipated three potential outcomes:

1. Positive Correlation: A developer contributing in multiple programming languages is more productive, possibly due to his or her ability to draw from multiple programming paradigms. For example, software developers working in a functional language such as Lisp arguably approach a problem differently than those writing in a purely object-oriented language such as Java.

[1]The log operation is undefined at zero; thus, languages with a $p_i = 0$ must be excluded from the calculation. As a result, for all values $c > 1$ the minimum language entropy of 0 occurs in the limit as some p_i approaches 1.

2. Negative Correlation: A developer contributing in multiple languages is less productive, possibly as a consequence of the added burden required to concurrently maintain skills in multiple programming languages.

3. No Correlation: A developer's productivity is independent of language fragmentation.

In this paper, we provide evidence of a relationship between language fragmentation and the problem-solving abilities of developers by demonstrating a significant *negative* correlation between language entropy and programmer productivity within the SourceForge community.

3.5 Data

The data set used in this study was previously collected for a separate, but related work. It was originally extracted from the August 2006 SourceForge Research Archive (SFRA). For a detailed discussion of the data source, including summary statistics and collection tools/processes, see work by Delorey, Knutson, and MacLean [21].

3.5.1 Description of the Data Set

The data set is composed of *all* SourceForge projects that match the following four criteria: 1) the project is open source; 2) the project utilized CVS for revision control; 3) the project was under active development as of August 2006; 4) the project was in a Production/Stable or Maintenance stage. The data set includes nearly 10,000 projects with contributions from more than 23,000 authors who collectively made in excess of 26,000,000 revisions to roughly 7,250,000 files [21].

A study by Delorey, Knutson, and Chun [19] identified more than 19,000 unique file extensions in the data set, representing 107 programming languages. The study also noted that 10 of those 107 languages are used in 89% of the projects, by 92% of the developers, and account for 98% of the files, 98% of the revisions, and 99% of the lines of code in the data set. Table 3.2 shows the 10 most popular languages with rankings. Delorey et al. ranked the languages based on the following five factors: 1) total number of projects using the language;

	Project Rank	Author Rank	File Rank	Revision Rank	LOC Rank	Final Rank
C	1	1	2	2	1	1
Java	2	2	1	1	2	2
C++	4	3	4	4	3	3
PHP	5	4	3	3	4	4
Python	7	7	5	5	5	5
Perl	3	5	9	9	6	6
JavaScript	6	6	6	8	10	7
C#	9	9	7	6	7	8
Pascal	8	10	8	7	8	9
Tcl	11	8	10	10	9	10

Table 3.2: Top ten programming languages by popularity rankings (account for 99% of the lines of code in the data set)

2) total number of developers writing in the language; 3) total number of files written in the language; 4) total number of revisions to files written in the language; and 5) total number of lines written in the language.

3.5.2 Producing a Data Sample

Because our analysis was computationally demanding, we extracted from the initial data set a random sample of 500 developers together with descriptive details of all revisions that those developers made since the inception of the projects on which they worked (for the purposes of this study, a *developer* is an individual who contributed at least one line of code in at least one revision to a source file). A sample size of 500 provides more than sufficient statistical precision to identify any practically significant relationships. This intuition is validated in the results by the extremely low p-values.

After sampling the data set, we condensed the sample by totaling the lines of code contributed by each developer for each month in which a developer made at least one code submission. Finally, we calculated the language entropy per month for each developer. Note that months in which a developer did not contribute are discarded because the language entropy metric is undefined for zero lines of code.

Inactive Months

Ignoring a developer's "inactive" months is reasonable for this study since we are more interested in the existence of a relationship between lines of code production and language entropy than we are in the actual magnitude of that relationship. However, our model does assume that the code was written in the month in which it was committed. Therefore, *months without submissions represent a confounding factor in this study.*

Filtering the Data

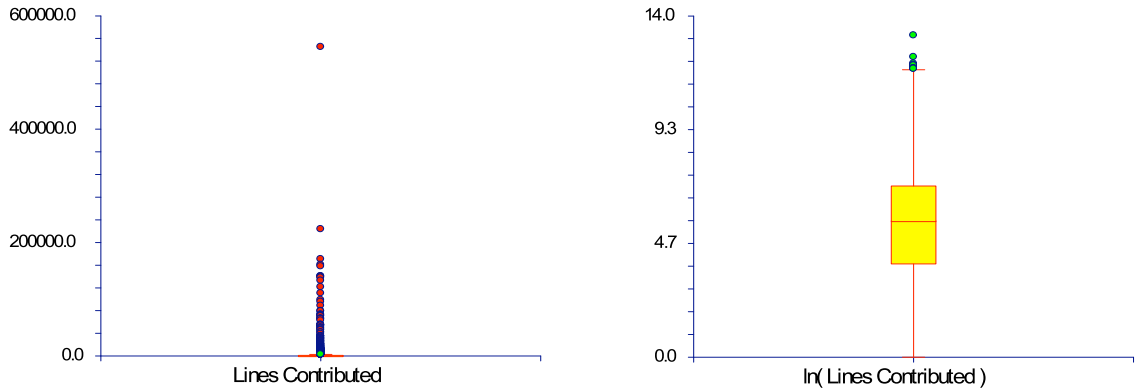
To help account for multi-month code submissions, as well as the six factors identified by Delorey et al. [19]—migration, dead file restoration, multi-project files, gatekeepers, batch commits, and automatic code generation—we applied filters to the data sample. However, analyses of the filtered and unfiltered data produced statistically indistinguishable results, suggesting that the data is insensitive to outliers. Therefore, *we report our results from the more robust, unfiltered data sample.*

For completeness, however, we describe the filtering technique: To filter the data, we 1) removed all data points of developers who submitted more than 5,000 LOC during at least three separate months, and 2) removed all data points for which a month's submission was greater than 5,000 LOC. The first filter was intended to remove project gatekeepers who submitted code on behalf of other developers. If a developer was suspected of being a gatekeeper, all of his/her contributions were excluded. The second filter was designed to remove significant quantities of auto-generated code.

We feel that these two filters are sufficient on the grounds cited by Delorey et al. [19], in which the authors controlled for outliers by capping the annual developer contribution at 80,000 LOC. Our limit of 5,000 LOC per month resulted in a maximum possible annual contribution of 60,000 LOC per developer—a bit more conservative.

3.6 Analysis

In this section, we first analyze the data sample for 500 randomly selected developers. We then select a statistical model appropriate for both the question of interest and the data (see Sections 3.4 and 3.5, respectively). We conclude this section by justifying and validating the selected model.



(a) Normal scale: Indicates significant outliers (notice the numerous individually plotted data points), a skewed distribution (so much so that the box plot is flattened into the x-axis), and a large data range (see the y-axis).

(b) Natural log scale: Indicates minimal outliers, the distribution is approximately normal, and the range is controlled.

Figure 3.2: Box plots of Lines Contributed

3.6.1 Transforming the Data

Figure 3.2(a) shows a box plot of the lines contributed. Three threats to the assumptions of a linear regression model are clearly visible: significant outliers, a skewed distribution, and a large data range. We adjust for all three issues by applying a natural log transformation. Notice in figure 3.2(b) that outliers are minimized, the distribution is approximately normal, and the range is controlled.[2]

[2]Statistical models assume certain characteristics about data. Sometimes data must be transformed before it can be accurately analyzed. However, interpretation of the results must reflect the transformation. In this study, for instance, the slope of the regression line must be interpreted as a multiplicative factor since a logarithmic transformation has been applied to the dependent variable.

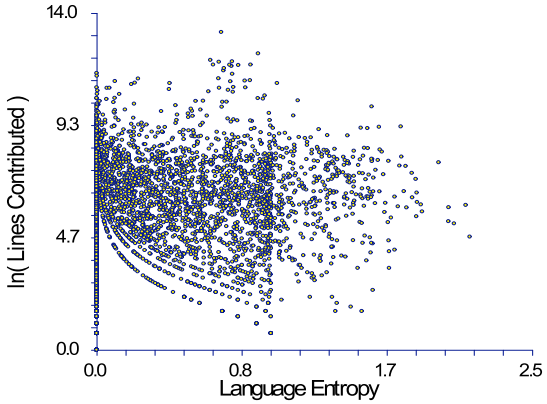


Figure 3.3: Plot of ln(Lines Contributed) vs. Language Entropy

Entropy Range	**Data Points**
$E(S) = 0$	2,394
$0 < E(S) \leq 1$	2,177
$1 < E(S)$	385
Total Data Points:	4,956

Table 3.3: Distribution of data points by Language Entropy

3.6.2 Selecting a Statistical Model

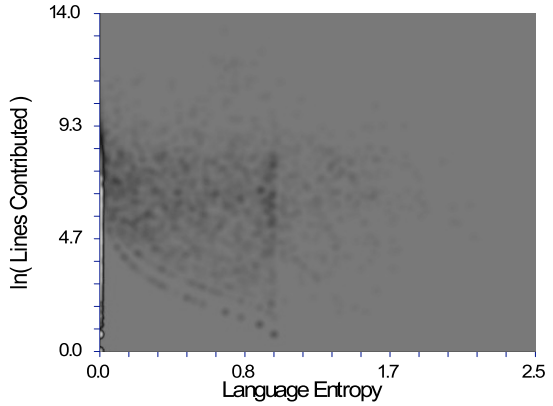
Figure 3.3 displays a plot of lines contributed (on the natural log scale) versus language entropy, in which each point on the graph represents one month of work for one developer. First, be aware that the volume and distribution of data points (see Table 3.3) is masked by crowding, which causes points to be plotted over other points. In total, nearly 5,000 points are plotted, of which approximately 48% lie on the y-axis at the entropy value of zero. Thus, nearly half the data consists of months in which developers submitted code in only one language. The distribution of the data points with respect to language entropy is consistent with the findings of Delorey, Knutson, and Giraud-Carrier who, for the same SourceForge data set, report that approximately 70% of developers write in a single language per year [20].

The relative density of the data is much easier to see in Figure 3.4. Density maps 3.4(a) and 3.4(b) confirm that the greatest density occurs on the y-axis. In fact, the data at entropy zero is so dense that it washes out the rest of the data. Density maps 3.4(c) and

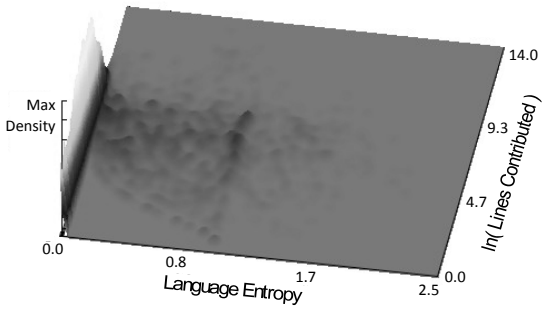
3.4(d) increase the contrast by calculating the densities for only the data with entropy values greater than zero.

Since this study intends to show a significant relationship between language entropy and lines contributed, we must demonstrate both a significant correlation between the two metrics and a reasonable variance in the data. The data plot and density maps, however, show a large spread in the data, indicating considerable variance. For the non-zero-entropy data (not on the y-axis), there does not appear to be a significant correlation between language entropy and lines contributed. However, the variance in the data is consistent with numerous studies in which the authors report large variability in programmer productivity (e.g., [42], [36], [22] and [17]). Thus, we do not expect to find consistent results *across* developers when examining productivity-related metrics. In this study we are interested in (and expect to find) a correlation *within* developers. Therefore, we use a *random coefficients model* to group the data by developer. Because this mixed model accounts for the non-independence of the data, it allows us to analyze trends within developers, as well as to combine all 500 analyses into a result that is representative of the SourceForge community.

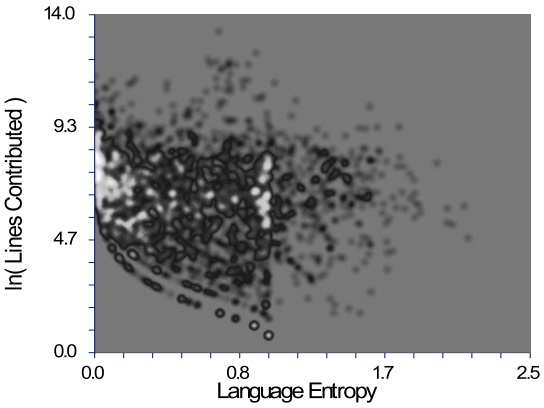
Further, because the distribution of the data is considerably different during months in which developers contributed code in only one language (zero entropy), versus months in which they contributed code in more than one language (entropy greater than zero), it would be inappropriate to apply a single regression line to the full range of the data. A *random coefficients model* solves this problem by allowing us to estimate a mean for the group at zero entropy, while fitting a regression line to the rest of the data. The two groups could be analyzed separately, but fitting them under one model allows us to pool the data when computing the error terms, which results in tighter confidence intervals and a more efficient analysis. Thus, our model estimates three parameters: 1) the mean of the data at zero entropy, 2) the slope of a regression line fit to the non-zero-entropy data, and 3) the intercept of the regression line.



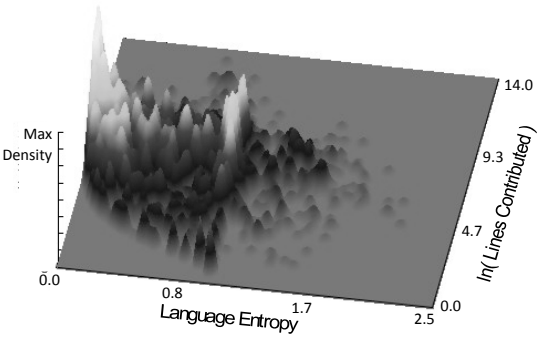
(a) Density *heat* map; data at $E(S) = 0$ dominates



(b) Density *height* map; data at $E(S) = 0$ dominates



(c) Density *heat* map; limit $E(S) > 0$

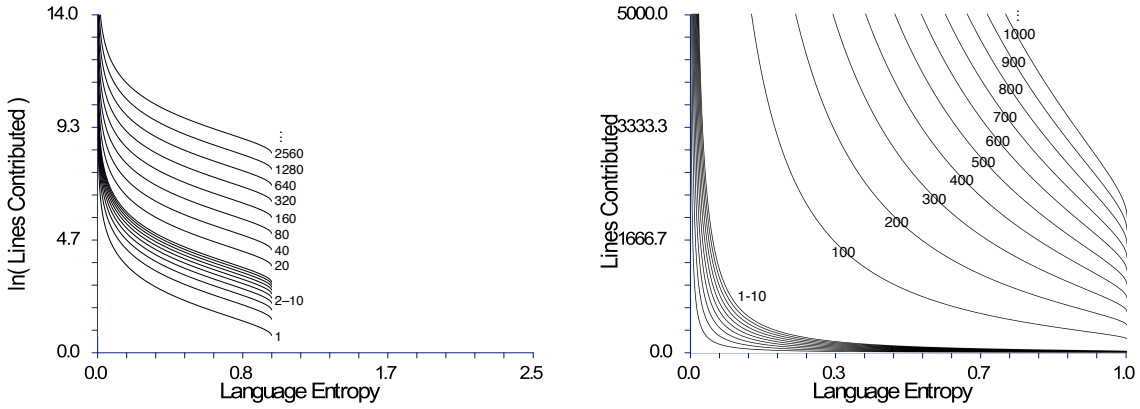


(d) Density *height* map; limit $E(S) > 0$

Figure 3.4: Relative density maps of ln(Lines Contributed) vs. Language Entropy

3.6.3 Adjusting for Serial Correlation

Another concern is the potential for serial correlation, which may occur when measurements are taken over time. Estimating the mean of serially-correlated data requires statistical adjustment in order to produce accurate results. The data sample in this study contains an average of eight months of measurements per developer, which is insufficient to confidently identify a serial correlation [44]. However, to be conservative we assume that serial correlation exists in the data and adjust for it in our analysis.



(a) Log scale

(b) Normal scale

Figure 3.5: Entropy bands for the two-language case (labeled by equivalence class from the axes)

3.6.4 Banding in the Data

The scatter plot in Figure 3.3 reveals a pattern of curving lines at the bottom of the point cloud between zero and one entropy. This banding pattern is due to the interplay between the metrics for language entropy and lines contributed. Specifically, the two metrics partition the data points into equivalence classes, one for each band on the graph. Figure 3.5(a) shows a graph of the equivalence classes on the log scale for the two-language case. Data points in the first equivalence class (forming the band closest to the x-axis) correspond to monthly contributions in which all but one line was written in the same language. Data points in the second equivalence class correspond to monthly contributions in which all but two lines were written in the same language. Notice that for each equivalence class, as the total lines contributed increases, the entropy score approaches zero. Entropy bands for three or more languages look similar to the two-language case, except that they extend to their respective maximum entropy values (refer back to Table 3.1).

Figure 3.5(a) also demonstrates that as the equivalence classes progress in the positive y-direction they grow exponentially closer together. By the fourth equivalence class the bands visibly blend on the graph. Thus, even though the bands are discrete in the y-direction, the space between them quickly becomes negligible.

Parameter	Estimate	Lower 95% CL	Upper 95% CL	p-value	Standard Error	DF
zeroEntropyGroupMean	4.0678	3.9262	4.2094	¡.0001	0.07209	499
nonZeroEntropyGroupDiff	2.1983	2.0152	2.3814	¡.0001	0.09300	259
nonZeroEntropyGroupSlope	-0.5072	-0.7281	-0.2863	¡.0001	0.11210	236

Table 3.4: Model parameter estimates on the log scale

Impact on Regression Coefficients

The banding pattern demonstrates that the discrete range of the LOC metric restricts the area of the graph into which data may fall. For the range of the data in Figure 3.3, the restriction appears significant, which brings into question the regression model previously discussed. Specifically, since the model assumes that the domain and range of the data are continuous, will it yield inaccurate results due to the non-continuous space into which the data are mapped by the metrics? It appears plausible from Figure 3.5(a) that the restricted area at the origin and/or the slope of the bands may cause the slope of a regression line to be inaccurately negative.

One method for validating the regression model is to test it on data for which the correlation between language entropy and lines contributed is known. Therefore, we produce an artificial data sample for the two-language case such that no correlation exists between language entropy and lines contributed. We generate our artificial sample by replacing all y-values in the real sample with random values. Each random y-value is selected from the range of all possible values that could produce the corresponding entropy score. In addition to limiting the sample to the two-language case, we also cap the maximum lines contributed at 5,000, a reasonable upper bound for one month's work for a single developer (see Section 3.5.2). Additionally, this limit increases the impact of the restricted area on the regression line. Applying these limits, the artificial sample incorporates 92.1% of the data points from the real sample. Further, the artificial sample retains many of the characteristics of the real sample (e.g., developer groupings).

Running the selected regression analysis on the artificial, non-correlated sample demonstrates that the shape of the space into which the data is mapped by the metrics does

not appreciably affect the model's slope parameter. For our artificial sample, the analysis results in a small negative slope that is not statistically distinguishable from zero (two-sided p-value of 0.50). Consequently, *any significant negative (or positive) slope found in the real data should indicate a true correlation between language entropy and lines contributed.*

This result is due to the fact that on the normal scale the restricted area at the origin is actually negligible for the range of the data (see Figure 3.5(b)). Logging the dependent variable does not compromise the analysis because the compression ratio of the log transformation increases exponentially as its argument increases linearly. In effect, the transformation's amplification of the low-range data is counteracted by the way it compresses the high-range data more significantly, causing the analysis to appropriately place greater weight on the high range.

The second parameter, that estimates the mean for the data at zero entropy, is also unaffected by the data mapping. The analysis of the artificial, non-correlated data yields a parameter estimate of 2,502 for the mean of the data at zero entropy (two-sided p-value less than 0.0001), as expected for data randomly selected from the range 1 to 5,000. Although the p-value is extremely low (because the sample size is large), a two-line deviation from the median of the range is not practically significant. Thus, *any practically significant deviation from the mean for the zero-entropy data would indicate a non-random effect.*

Although the mean for the zero-entropy data and the slope of the regression line for the non-zero-entropy data are not affected by the data mapping, the intercept of the regression line *is* affected. The analysis of the artificial, non-correlated data yields an intercept of 3,311 LOC—809 lines above the median of the data range (2,500 LOC). The positive shift in the intercept of the regression line is another artifact of the interplay between the metrics, which results in a mapping of the data into a space with a density gradient that increases radially from the origin (see Figure 3.5(b)). Thus even before taking the natural log of the dependent variable, the higher-range data is denser, artificially pulling up the intercept of the regression line. The regression model accounts for the density shift due to the log transformation, but

not for the gradient introduced by the metrics. Thus, *finding a positive difference in the real data sample between the intercept of the regression line and the mean of the zero-entropy data may not indicate a real difference between the two groups.*

3.6.5 Boundary at Entropy Value of 1.0

The data exhibit a vertical boundary at the entropy value of 1.0 (refer back to Figure 3.3). This pattern is a consequence of the distribution of the data points. Delorey, Knutson, and Giraud-Carrier found in their analysis of the SourceForge data set that only 10% of developers use more than two languages per year [20]. As a result, we expect the data beyond two languages to be sparse. Since entropy values greater than 1.0 can only belong to the case of three or more languages, the boundary at the entropy value of 1.0 is simply an artifact of the shift in data point density around the maximum entropy value for the two-language case (as is evident from the density maps in Figure 3.4).

3.7 Results

Table 3.4 shows estimates (on the natural log scale) of the model parameters, with confidence intervals and two-sided p-values. All three parameters are statistically significant with p-values less than 0.0001. Such small p-values allow us to confidently conclude that the relationship between language entropy and lines contributed is *not* due to random chance. The low error terms (which result in narrow confidence intervals around the parameter estimates) give us confidence that our sample size is sufficient to accurately estimate the population variance. Further, since the data sample was randomly selected (as described in section 3.5.1), we can conclude that the observed patterns characterize the SourceForge community. However, since this is an observational study, we cannot infer causality. Therefore, the remainder of the discussion of results describes the magnitude of the observed relationship between language entropy and lines contributed.

In Table 3.4, the *zeroEntropyGroupMean* is an estimate of the mean of the data points at zero language entropy (the zero group, or ZG). The *nonZeroEntropyGroupDiff* represents the estimated difference between the ZG mean and the intercept of the regression line for the non-zero-entropy data (the non-zero group, or NZG). The very low p-value for this parameter would normally indicate that the ZG mean is significantly different from the trend in the NZG. However, as discussed in Section 3.6.4, a positive difference between the intercept of the regression line and the estimate of the mean at entropy zero may be nothing more than an artifact of the metrics. Adding the first two parameter estimates gives the estimate for the intercept of the NZG regression line (6.2661). The third parameter, *nonZeroEntropyGroupSlope*, represents the slope of the NZG regression line, which is negatively correlated with language entropy.

The magnitudes of these parameter estimates make more sense on the original scale. However, the back-transformed estimates must be reinterpreted because the analysis is performed on log-transformed data. Specifically, the ZG mean and the intercept of the NZG regression line both represent medians on the original scale. Further, the slope of the NZG regression line becomes a multiplicative factor, which means that an increase in language entropy results in a multiplicative decrease in lines contributed. Equations 3.4 and 3.5 show the back-transformed model, and Figure 3.6 shows the model graphed on the normal scale.

$$\begin{aligned} ZG_{median} &= e^{4.0678} \\ &= 58.4 \end{aligned} \tag{3.4}$$

$$\begin{aligned} NZG &= e^{(4.0678+2.1983)} e^{-0.5072x} \\ &= 526.4(e^{-0.5x}) \end{aligned} \tag{3.5}$$

For months in which a developer submits code in one language (ZG), the developer contributes, on average, 58 LOC (95% confidence interval from 51 to 67 LOC). However,

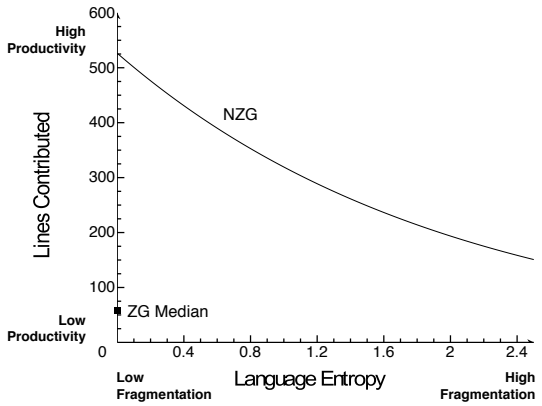


Figure 3.6: Best-fit model on the normal scale

extrapolating the trend in the NZG, which represents months in which developers submitted code in more than one language, one would expect the ZG median to be 526 LOC—a significant difference. Thus, the best-fit model for the data indicates that during months in which a developer contributes code in only one language, the developer also tends to contribute significantly less code than during months in which he or she contributes in more than one language.

However, taking into account the fact that the metrics artificially inflate the intercept of the regression line in our analysis (see Section 3.6.4), the positive difference between the intercept of the regression line and the mean of the zero-entropy data may not be a real effect. Further, this difference considers both highly and marginally active developers equally. The marginally active developers, who make only a few small contributions (and for whom a productivity increase is relatively uninteresting), are likely pulling down the ZG median. In particular, when a developer writes only a small amount of code it is more likely that the developer will write in a single language. Removing marginally active developers, therefore, should remove more data points from those on the y-axis than from the rest of the graph, which would reduce the difference between the two groups.

For months in which a developer submits code in more than one language, the developer's monthly contributions decrease by an estimated 4.9% for each 0.1 unit increase in language entropy (95% confidence interval from 2.8% to 7.0%). For a 1.0 unit increase

in language entropy (e.g., writing equally in two languages versus writing predominantly in one language), a developer's monthly contribution drops by approximately 39.8% on average (95% confidence interval from 24.9% to 51.7%).

Thus, in answer to the central question—**What is the relationship between programmer productivity and the concurrent use of multiple programming languages?**—*for a developer who programs in multiple languages, it appears that he or she is most productive when language fragmentation is minimal (i.e., the developer programs predominately in a single language).*

3.8 Conclusions

The primary objective of this study was to test the relationship between programming language fragmentation and developer productivity in the SourceForge community. The results of the study demonstrate a significant negative correlation between language entropy and the size of developer contributions. Since the data was randomly selected, we can make inferences to the general SourceForge community for those developers who worked on open-source, Production/Stable or Maintenance projects using CVS from 1995 through August 2006. Specifically, for SourceForge developers writing in multiple languages, we can infer with high confidence that writing evenly across languages negatively impacts the size of monthly code contributions. However, because our study is observational, we cannot infer that the differences in language entropy caused the observed variation in productivity. Nevertheless, the results open up avenues of research for investigating the relationship and possible effects of multi-language development on productivity.

We also have high statistical confidence that, for SourceForge developers writing in a single language, the average monthly contribution is about 58 LOC. However, since our sample includes minimally active developers, this estimate is likely too low for full-time, professional developers. Although we can generalize this result to the SourceForge community, conclusions about the more interesting group of active developers are somewhat suspect.

Additionally, without further analysis we cannot make conclusions about the productivity difference between writing in a single language versus multiple languages. Applying our analysis tools to the non-correlated data clearly demonstrates that the tools are unable to accurately differentiate these two groups.

3.9 Limitations

In the following subsections we identify several limitations of this study.

3.9.1 Inferences

Our inferences are limited to developers on SourceForge. Therefore, we cannot make general conclusions about other software development environments. Also, the SourceForge archive obscures certain information about developers (such as the identity of gatekeepers). These issues would be best addressed through replication of results in other development environments.

This study also does not confirm causality inferences. To understand the cause of the observed relationship between language entropy and lines contributed, we would need to run controlled, randomized experiments (see Section 3.10.1).

3.9.2 Non-Contributing Months

The developers in our data set did not always contribute to projects in contiguous months. For example, a developer might contribute changes in April, skip May, and contribute again in June. For the purposes of this study we assumed that developers submitted contributions in the same months in which those contributions were written. We took steps to help ensure our assumption (see Section 3.5.2 and 3.5.2). However, the data likely still contain some instances that violate the assumption, for which we have not been able to control. Although we believe the impact of such instances to be minimal, the extent of their impact on the study results is unknown.

3.9.3 Productivity Measure

Despite its utility in this study, the LOC/month metric is only one of many programmer productivity metrics. Further studies could extend this analysis to other productivity models. Additionally, our productivity model did not account for differing levels of programming language verbosity (e.g., Perl versus C). In a future study we may be able to normalize the data using average commit size as an estimate of language verbosity.

3.9.4 Marginally Active Developers

Developers who make only small contributions per month may bias the analysis results. First, these marginally active developers are probably less likely to write in multiple languages during a given month. In this case, filtering them out could reduce the disparity between the zero- and non-zero-entropy groups (especially considering the power law trends found by Healy and Schussman in SourceForge data [29]). Further, the estimated contribution averages for the active developer group are much less likely to suffer from sampling error (not to mention that the active group is more interesting to study from a productivity standpoint). Thus, it would be valuable to repeat this study with only "active" developers.

3.10 Future Work

In this section we outline avenues for future research.

3.10.1 Establishing Causality

This study demonstrates a correlation between language entropy and the size of developer contributions within the population of SourceForge developers. To understand the cause of the observed relationship we need to run controlled randomized experiments. We believe that such efforts, in combination with corporate case studies (as described in Section 3.10.2), would provide meaningful results from which practitioners could make better-informed decisions regarding project-developer assignments and the adoption of new languages and frameworks.

3.10.2 Corporate Case Studies

Running an impact analysis of language entropy utilizing data from industry projects would allow us to expand our inferences into the corporate domain, at which point we could ask a number of important questions, including:

- If my company is maintaining a large code base in COBOL, how will my developers' productivity be affected by an additional project in Java?[3]
- My company already supports products in different languages. Will my developers be more productive if I assign each one to a specific language, as opposed to spreading them across languages?

3.10.3 Paradigm Relationships

Many of the languages in our study seem to cluster by paradigm (for example, object-oriented languages such as Java, C++, and C#). Switching between programming languages that share a common paradigm may not be as cognitively difficult as switching between languages from different paradigms. We expect changes in language fragmentation to affect a programmer working within a single paradigm less than one working across multiple paradigms.

3.10.4 Commonly Grouped Languages

In this study, we examine the relationship between language entropy and productivity across all languages. However, some languages are commonly used together (e.g., many web projects are based on Java, JavaScript, and HTML). Is the cognitive burden of context switching between languages reduced for developers who work across a set of commonly grouped languages? What about the burden of maintaining skills in multiple languages?

[3]Lest the reader dismiss this example as unrealistic, the scenario is taken from an actual corporate project, the thrust of which is a massive migration of application software from COBOL to Java.

3.10.5 Language Fragmentation as a Productivity Measure

To better understand the relationship between language fragmentation and other productivity metrics, we need to determine whether language fragmentation provides new information beyond the metrics already presented in the literature. If shown to be complementary, language fragmentation can be incorporated into more complex productivity and cost-estimation models [14].

3.11 Acknowledgements

We are grateful to reviewers and attendees of the 4th Workshop on Public Data about Software Development (WoPDaSD 2009) in Skovde, Sweden for valuable feedback on early versions of this work. We are also grateful to Christian Bird for his insightful comments on a draft of this paper.

Chapter 4

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.

4.1 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" [33]. 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.

4.1.1 SourceForge as a Data Source

Over 100 researchers use the SourceForge Research Data Archive (SRDA)[5] 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 [30]. In section 4.2 we discuss our insights into the limitations of SourceForge for studying certain project attributes.

4.1.2 Language Fragmentation

The targeted study explored the correlation between language fragmentation and programmer productivity. Fragmentation is measured by language entropy (originally defined in [32]), 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.

4.1.3 Data Set

For this study and [33] 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 [21].

4.1.4 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.

4.2 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.

4.2.1 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).

4.2.2 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

$$E(S) = -\sum_{i=1}^{c}(p_i \times \log_2 p_i) \tag{4.1}$$

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

$$\log_2 c \tag{4.2}$$

[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.

4.2.3 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 4.1 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.

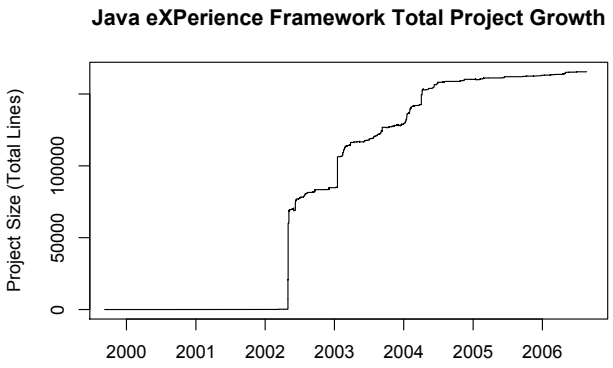


Figure 4.1: Growth of the Java eXPerience FrameWork over time.

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 4.2). 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.

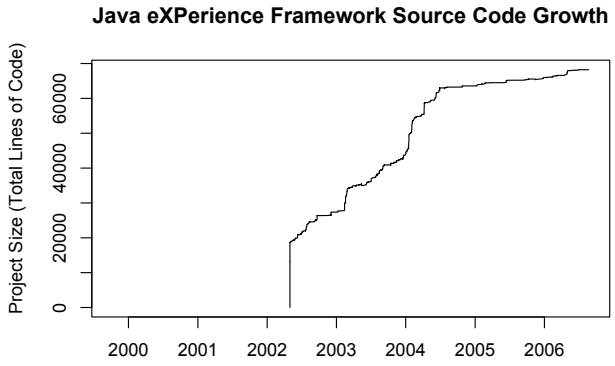


Figure 4.2: Growth of the Java eXPerience FrameWork over time.

4.2.4 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.

4.2.5 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 4.2 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 4.3. Unfortunately, without further data we can only make educated guesses.

4.2.6 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.

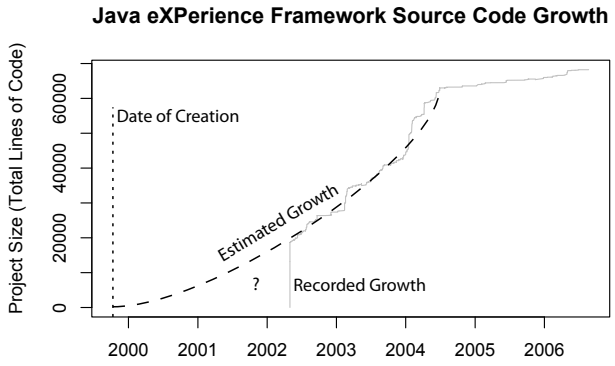


Figure 4.3: Growth of the Java eXPerience FrameWork over time with estimate of actual growth.

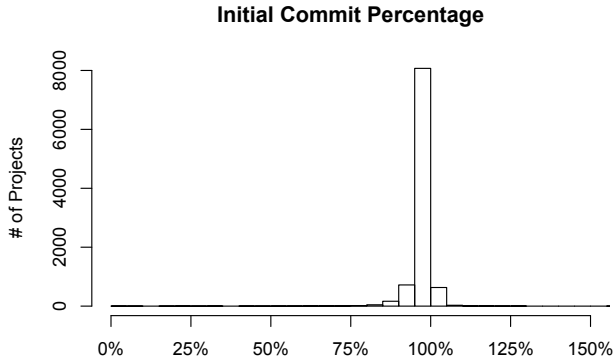
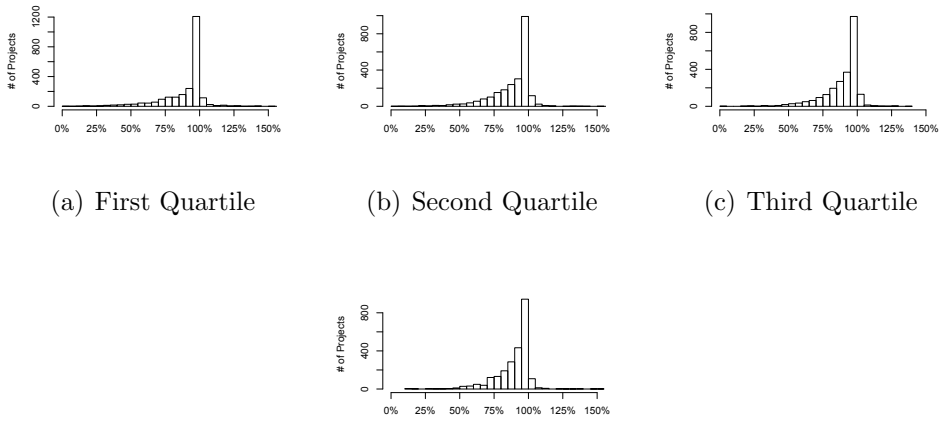


Figure 4.4: Percentage of the *Project Size* explained by initial commits.

4.2.7 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 4.4). 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 5.6).

[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 4.5: Percentage of the *Project Size* explained by initial commits by *Project Size* quartile.

4.2.8 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 4.1 is just shy of 50% of the total project size

4.3 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 [33]. 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.

4.3.1 Marginally Active Developers

Krein, et. al. [33] 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 4.4) 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.

4.3.2 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 4.2, this approach is likely insufficient to mitigate the potentially induced error.

Figure 4.3 shows a time period of two and a half years during which development occurred, but for which data is entirely missing. In [33] 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 [33] 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.

4.3.3 Author Project Size Bridging

Analysis of author contributions reveals that authors don't often bridge between project sizes (see Table 4.1). 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 4.6 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.

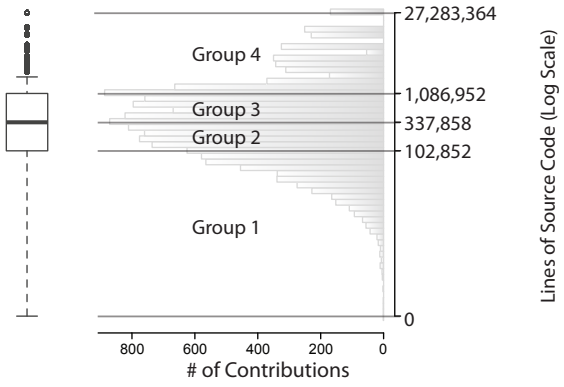


Figure 4.6: Project size groups.

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 4.1: Author bridging, or lack thereof, between *Project Size* groups (see Section 4.3.3).

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 4.2). 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		Combined	
Group	Authors	Percentage	Authors	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 4.2: Author bridging, or lack thereof, between *Project Size* groups for authors who contribute to multiple projects.

4.4 Limitations in the Original Study

In [33], 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 [33] 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.

4.5 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.

4.5.1 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.

4.5.2 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 4.7. 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.

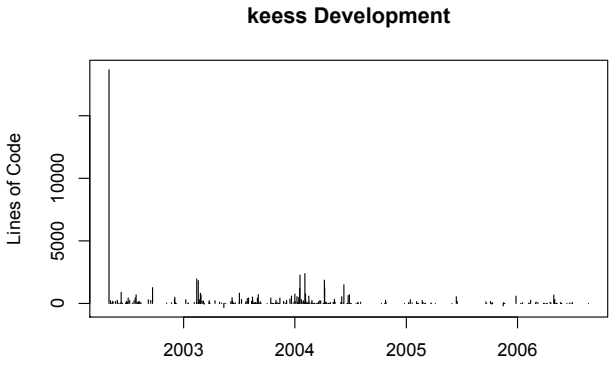


Figure 4.7: Development behavior of 'keess.'

4.5.3 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.

4.5.4 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 4.4, 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.

4.5.5 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 5

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.

5.1 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. [40, 18]

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 [14]. 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., [10, 19, 28, 32, 39, 50, 53]). 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 [45], recent studies suggest that the two may not be as different as originally thought [11, 23]. Further, as one of the largest OSS hubs, SourceForge hosts thousands of projects—providing extensive data on thousands of mature projects [20]. 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 [32, 33]. 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.*

5.2 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

[1]The number of subscribers to the SRDA (SourceForge Research Data Archive) currently exceeds 100 [5]. The actual number of researchers engaging SourceForge data is likely several times that.

9,997 Production/Stable or Maintenance phase projects stored in CVS on SourceForge and extracted in October of 2006 [19].

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.

5.2.2 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 5.1). In our analysis we group commits into days to identify cliff walls programmatically.

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 5.2 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

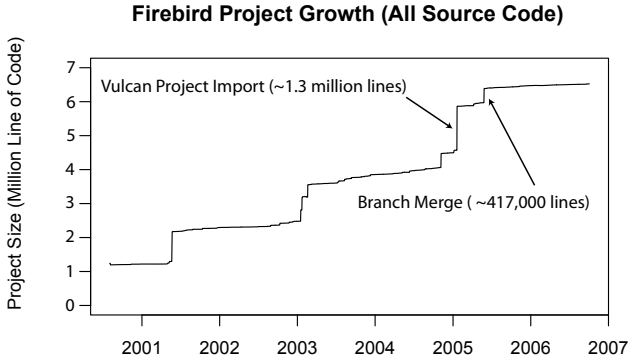


Figure 5.1: Growth of Firebird over time.

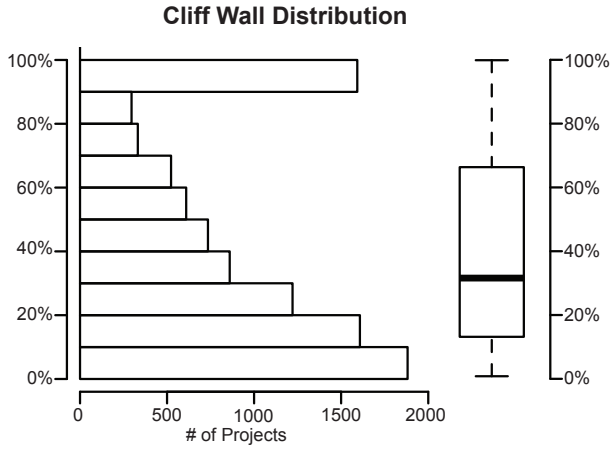


Figure 5.2: Distribution of projects by largest cliff walls. One outlier has been removed.[2]

$0 - 10\%$ 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 5.1 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 5.3). 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%.

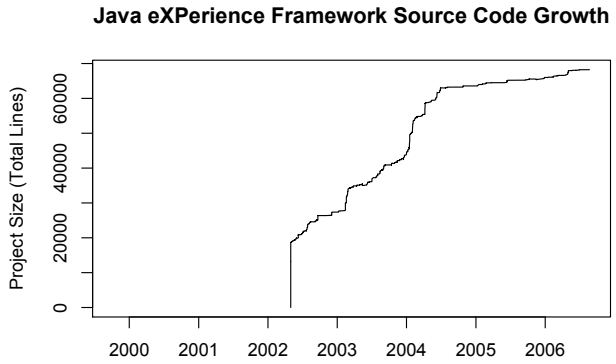


Figure 5.3: Growth of the Java eXPerience FrameWork over time.

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 5.3 we discuss four of those reasons.

5.2.3 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 5.4 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 5.5; note the log scale on the y-axis). However, this

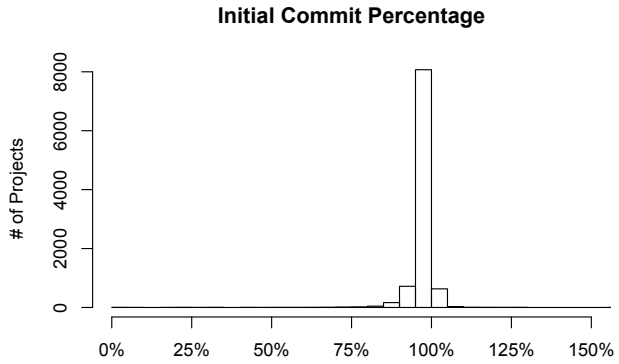


Figure 5.4: Distribution of projects by Initial Commit Percentage.

distribution holds, with small variation, regardless of project size (see Figure 5.6). High ICP indicates that revisionary changes to existing files constitute a small percentage of project growth.

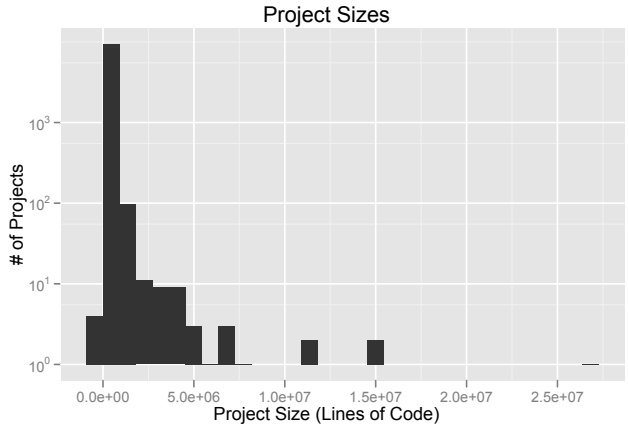
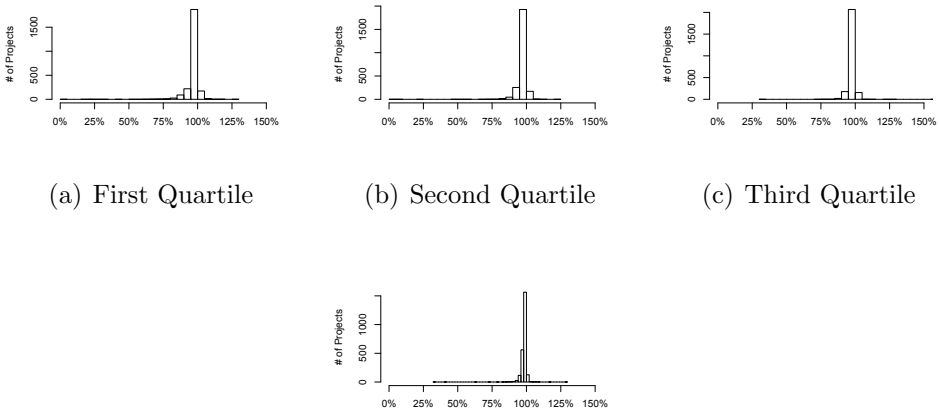


Figure 5.5: Project sizes.

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 5.3.



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

(d) Fourth Quartile

Figure 5.6: Distribution of project by Initial Commit Percentage discretized by project size quartile.

5.3 Reasons for Problems

Although there are many possible causes for the anomalies mentioned in Section 5.2, 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.

5.3.1 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" [30].

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 5.3.4) 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".

5.3.2 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 [37]). 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 [37]. 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 [48]. 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.

5.3.3 Project Imports

In Figure 5.1 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.

5.3.4 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 [54].

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 5.1 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.

5.4 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.

5.4.1 Identify Merges

In Section 5.3.4 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.

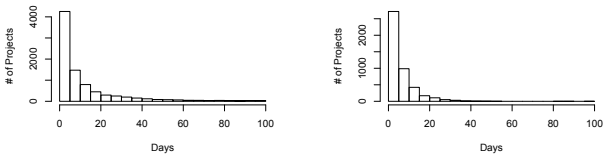
Zimmermann et al, suggest a very simple approach to identifying merge transactions wherein researchers manually examine each transaction for which the log message contains the word "merge" and determine if the transaction is indeed the merge of a branch [54]. There are drawbacks to this approach. First, 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 [26]. 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.

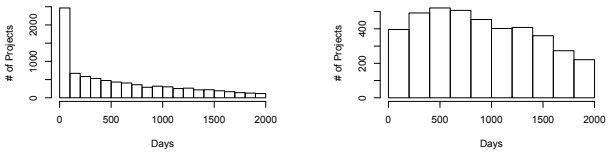
5.4.2 Author Behavior

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 5.7 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 5.7: Distribution of projects by frequency of author commits.



(a) All Projects (b) ¿ 40 commits

Figure 5.8: Distribution of projects by project life span: the time between the first and the last commit in a project.

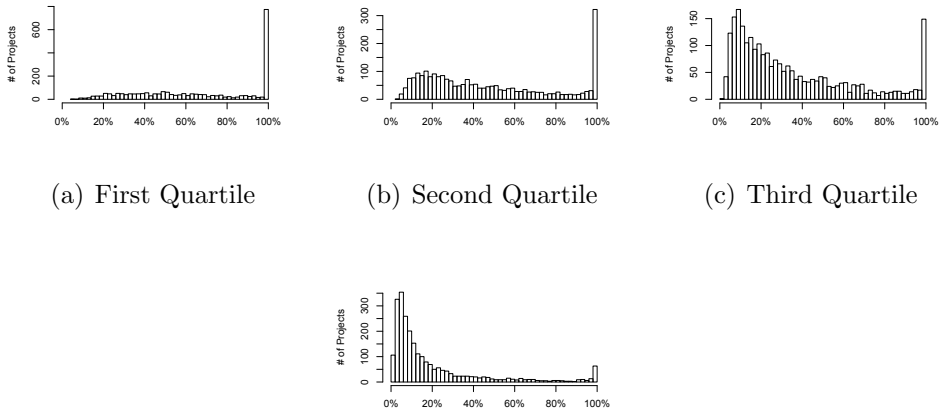
to show that the graphic is not overly biased by small projects that are completed quickly. There appear to be plenty of projects that satisfy a high *frequency of commits* requirement. In Figure 5.8 we see that by limiting the data set to projects with more than 40 commits we also get rid of most of the short-lived projects.

5.4.3 Project Size

Small projects have a much higher occurrence of large cliff walls than large projects. Figure 5.9(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 5.9 have a spike at 100%. However, 5.9(b), 5.9(c), and 5.9(d) have successively greater area under the curve towards 0% (see Table 5.1 for quartiles). This

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

Table 5.1: Project Size Quartiles (Lines of Code)



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

(d) Fourth Quartile

Figure 5.9: Distribution of projects by largest cliff wall as a percentage of project size. See Section 5.2.2 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.

5.5 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 5.7 and 5.8 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.

References

[1] IEEE Standard for Software Productivity Metrics. *IEEE Std 1045-1992*, 1993.

[2] Call for Papers. *Proceedings of the International Workshop on Mining Software Repositories*. `http://msr.uwaterloo.ca/TSE04/CFP.pdf`, 2004.

[3] Call For Participation. *International Conference on Open Source Systems*. `http://oss2005.case.unibz.it/Resources/Calls/OSS2005CFP.txt`, 2005.

[4] 5th Workshop on Public Data about Software Development Call For Papers, 2010.

[5] Matthew Van Antwerp and Greg Madey. Advances in the SourceForge Research Data Archive (SRDA). In *Fourth International Conference on Open Source Systems*, Milan, Italy, September 2008.

[6] Juliana V. Baldo, Nina F. Dronkers, David Wilkins, Carl Ludy, Patricia Raskin, and Jiye Kim. Is Problem Solving Dependent on Language? *Brain and Language*, 92(3):240–250, 2005.

[7] Victor Basili, Scott Green, Oliver Laitenberger, Filippo Lanubile, Forrest Shull, Sivert Sørumgård, and Marvin Zelkowitz. The Empirical Investigation of Perspective-Based Reading. *Empirical Software Engineering*, 1(2):133–164, 1996.

[8] Ellen Bialystok and Shilpi Majumder. The Relationship Between Bilingualism and the Development of Cognitive Processes in Problem Solving. *Applied Psycholinguistics*, 19(1):69–85, 1998.

[9] Alessandro Bianchi, Danilo Caivano, Filippo Lanubile, and Giuseppe Visaggio. Evaluating Software Degradation through Entropy. volume 0, page 210, Los Alamitos, CA, USA, 2001. IEEE Computer Society.

[10] Christian Bird, Alex Gourley, Prem Devanbu, Anand Swaminathan, and Greta Hsu. Open Borders? Immigration in Open Source Projects. volume 0, page 6, Los Alamitos, CA, USA, 2007. IEEE Computer Society.

[11] 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.

[12] Barry Boehm. Managing Software Productivity and Reuse. *IEEE Computer*, 32(9):111–113, 1999.

[13] Barry Boehm, Terrence Gray, and Thomas Seewaldt. Prototyping Versus Specifying: A Multiproject Experiment. *IEEE Transactions on Software Engineering*, (3):290–303, 1984.

[14] Lionel Briand, Sandro Morasca, and Victor Basili. Defining and Validating Measures for Object-Based High-Level Design. volume 25, pages 722–743, Sep/Oct 1999.

[15] D.N. Card, F.E. McGarry, and G.T. Page. Evaluating Software Engineering Technologies. *IEEE Transactions on Software Engineering*, 13(7):845–851, 1987.

[16] Samuel Conte, Hubert Dunsmore, and Vincent Shen. *Software Engineering Metrics and Models*. Benjamin/Cummings Series in Software Engineering. Benjamin/Cummings Publishing Company, Inc., Menlo Park, California, USA, 1986.

[17] Bill Curtis. Substantiating Programmer Variability. *Proceedings of the IEEE*, 69(7):846, July 1981.

[18] Cleidson de Souza, Jon Froehlich, and Paul Dourish. Seeking the Source: Software Source Code as a Social and Technical Artifact. In *Proceedings of the 2005 International ACM SIGGROUP Conference on Supporting Group Work*, pages 197–206, New York, NY, USA, 2005. ACM.

[19] 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*, May 2007.

[20] 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*, June 2007.

[21] Daniel P. Delorey, Charles D. Knutson, and Alex MacLean. Studying Production Phase SourceForge Projects: A Case Study Using cvs2mysql and SFRA+. In *Second International Workshop on Public Data about Software Development*, June 2007.

[22] Tom DeMarco and Timothy Lister. Programmer Performance and the Effects of the Workplace. In *Proceedings of the 8th International Conference on Software Engineering*, pages 268–272, Los Alamitos, California, USA, 1985. IEEE Computer Society Press.

[23] Nicolas Ducheneaut. Socialization in an Open Source Software Community: A Socio-Technical Analysis. *Computer Supported Cooperative Work*, 14(4):323–368, 2005.

[24] Anne Smith Duncan. Software development productivity tools and metrics. In *Proceedings of the 10th International Conference on Software Engineering*, pages 41–48, Los Alamitos, CA, USA, 1988. IEEE Computer Society Press.

[25] Albert Endres and Dieter Rombach. *A Handbook of Software and Systems Engineering: Empirical Observations, Laws and Theories*, chapter 9, pages 190–192. Fraunhofer IESE Series on Software Engineering. Pearson Education Limited, Harlow, England, 2003.

[26] Michael Fischer, Martin Pinzger, and Harald Gall. Populating a Release History Database from Version Control and Bug Tracking Systems. In *Proceedings of the International Conference on Software Maintenance*, page 23, Washington, DC, USA, 2003. IEEE Computer Society.

[27] Warren Harrison. An Entropy-Based Measure of Software Complexity. *IEEE Transactions on Software Engineering*, 18(11):1025–1029, Nov 1992.

[28] 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.

[29] Kieran Healy and Alan Schussman. The Ecology of Open-Source Software Development. Technical report, University of Arizona, USA, January 2003.

[30] James Howison and Kevin Crowston. The Perils and Pitfalls of Mining SourceForge. In *Proceedings of the International Workshop on Mining Software Repositories*, pages 7–11, 2004.

[31] Capers Jones. *Software Assessments, Benchmarks, and Best Practices*. Addison-Wesley Longman Publishing Co., Inc., Boston, Massachusettes, USA, 2000.

[32] 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.

[33] 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*, volume 2, pages 41–61, June 2010.

[34] Alexander C. MacLean, Landon J. Pratt, Jonathan L. Krein, and Charles D. Knutson. Threats to Validity in Analysis of Language Fragmentation on SourceForge Data. In *Proceedings of the 1st International Workshop on Replication in Empirical Software Engineering Research*, May 2010.

[35] Alexander C. MacLean, Landon J. Pratt, Jonathan L. Krein, and Charles D. Knutson. Trends That Affect Temporal Analysis Using SourceForge Data. In *Proceedings of the 5th International Workshop on Public Data about Software Development*, June 2010.

[36] Katrina Maxwell, Luk Van Wassenhove, and Soumitra Dutta. Software Development Productivity of European Space, Military and Industrial Applications. *IEEE Transactions on Software Engineering*, 22(10):706–718, October 1996.

[37] Pam McDonald, Dan Strickland, and Charles 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.

[38] Audris Mockus. Amassing and Indexing a Large Sample of Version Control Systems: Towards the Census of Public Source Code History. In *6th IEEE International Working Conference on Mining Software Repositories, 2009*, pages 11–20. IEEE, 2009.

[39] 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.

[40] Kumiyo Nakakoji, Yasuhiro Yamamoto, Yoshiyuki Nishinaka, Kouichi Kishida, and Yunwen Ye. Evolution Patterns of Open-Source Software Systems and Communities. In *Proceedings of the International Workshop on Principles of Software Evolution*, pages 76–85, New York, NY, USA, 2002. ACM.

[41] Edward Nelson. Management Handbook for the Estimation of Computer Programming Costs. Technical report, Systems Development Corporation, 1966.

[42] Lutz Prechelt. The 28:1 Grant/Sackman Legend is Misleading, or: How Large Is Interpersonal Variation Really? Technical report, Universität Karlsruhe, Fakultät für Informatik, Germany, December 1999.

[43] Lutz Prechelt and Walter Tichy. A Controlled Experiment to Assess the Benefits of Procedure Argument Type Checking. *IEEE Transactions on Software Engineering*, 24(4):302–312, 1998.

[44] Fred L. Ramsey and Daniel W. Schafer. *The Statistical Sleuth: A Course in Methods of Data Analysis*, chapter 15, pages 436–455. Duxbury, Pacific Grove, California, USA, 2nd edition, 2002.

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

[46] Eric S. Raymond et al. *The Cathedral and the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary.* O'Reilly & Associates, Inc., 2001.

[47] Claude Shannon. A Mathematical Theory of Communication. *The Bell System Technical Journal*, 27:379–423, 623–656, Jul/Oct 1948.

[48] ShinjiUchida, Akito Monden, Naoki Ohsugi, Toshihiro Kamiya, Ken-Ichi Matsumoto, and Hideo Kudo. Software Analysis by Code Clones in Open Source Software. *The Journal of Computer Information Systems*, 45(3):1–11, 2005.

[49] Forrest Shull, Filippo Lanubile, and Victor Basili. Investigating Reading Techniques for Object-Oriented Framework Learning. volume 26, pages 1101–1118. IEEE, 2000.

[50] Alexander Tarvo. Mining Software History to Improve Software Maintenance Quality: A Case Study. *IEEE Software*, 26(1):34–40, 2009.

[51] Quinn C. Taylor, James E. Stevenson, Daniel P. Delorey, and Charles D. Knutson. Author Entropy: A Metric for Characterization of Software Authorship Patterns. In *3rd International Workshop on Public Data about Software Development*, September 2008.

[52] Piotr Tomaszewski and Lars Lundberg. Software Development Productivity on a New Platform: an Industrial Case Study. *Information and Software Technology*, 47(4):257–269, 2005.

[53] Jin Xu, Yongqin Gao, Scott Christley, and Gregory Madey. A Topological Analysis of the Open Souce Software Development Community. volume 7, Los Alamitos, CA, USA, 2005. IEEE Computer Society.

[54] Thomas Zimmermann, Peter Weißgerber, Stephan Diehl, and Andreas Zeller. Mining Version Histories to Guide Software Changes. *IEEE Transactions on Software Engineering*, 31(6):429–445, 2005.