2011-02-10

Programming Language Fragmentation and Developer Productivity: An Empirical Study

Jonathan L. Krein
Brigham Young University - Provo

Programming Language Fragmentation and Developer Productivity: An Empirical Study

Jonathan Leo Krein

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

Master of Science

Department of Computer Science

Brigham Young University

December 2011

ABSTRACT

Programming Language Fragmentation and Developer Productivity:

An Empirical Study

Jonathan Leo Krein

Department of Computer Science

Master of Science

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. Although language specialization arguably introduces benefits, the total impact of the resulting *language fragmentation* (working concurrently in multiple programming languages) 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 outweigh those benefits.

This thesis represents a first step toward understanding the relationship between language fragmentation and programmer productivity. We address that relationship within two different contexts: 1) the individual developer, and 2) the overall project. Using a data-centered approach, we 1) develop metrics for measuring productivity and language fragmentation, 2) select data suitable for calculating the needed metrics, 3) develop and validate statistical models that isolate the correlation between language fragmentation and *individual* programmer productivity, 4) develop additional methods to mitigate threats to validity within the developer context, and 5) explore limitations that need to be addressed in future work for effective analysis of language fragmentation within the project context using the SourceForge data set. Finally, we demonstrate that within the open source software development community, SourceForge, language fragmentation is negatively correlated with *individual* programmer productivity.

Keywords: software engineering, empirical, software metrics, programming languages, language fragmentation, productivity, problem solving, cognition, language entropy, open source, SourceForge

Contents

List of Figures

List of Tables

Chapter 1

Introduction

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

1.1 Linguistics and Problem Solving

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, pp. 240, 249]. 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 [10], which focused on the differences between mono- and bilingual children, found specific differences in the children's abilities to solve problems. The study results indicate that balanced bilinguals possess a greater degree of attention control, which is a necessary component of problem solving.

1.2 Language Fragmentation and Programmer Productivity

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. Specifically, are developers *more* or *less* productive at solving software problems when they implement their solutions in multiple programming languages—or does language fragmentation have no impact on developer cognition and performance? We anticipate one of three potential answers to this question:

1. Positive Effect: 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.
2. Negative Effect: 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 Effect: A developer's productivity is independent of language fragmentation.

In answering the question of interest, this work aims 1) to establish methods for studying language fragmentation generally and 2) to justify further exploration of the effects of language fragmentation on developer performance—particularly, in the corporate domain.

1.3 Applications of this Research

Studying the effects of language fragmentation on developer performance has the potential to affect both Software Engineering research and industry best practices. In general, discovering either a positive or negative *causal*[1] effect could produce improvements in: 1) programmer productivity models—and, consequently, in software cost-estimation models; 2)

[1] This is an observational study and, although suggestive, is insufficient to conclude causality.

the utilization of programming languages within specific projects; 3) the utilization of programming languages within software development organizations—particularly with respect to developer assignments and team organization; and 4) the design and use of *companion languages*.[2]

1.3.1 Software Engineering Research

Numerous Software Engineering journals, conferences, and workshops deal with subjects related to this research, including software metrics, the organizational and cognitive aspects of software development, open-source software communities and processes, and the use and design of programming languages.

One of the fundamental applications of this research within Software Engineering is to improve programmer productivity and software cost-estimation models. Programmer productivity has been studied for more than forty years, during which time researchers and practitioners have created numerous productivity metrics and models [44, 22, 36]. Any factor that could be shown to significantly impact productivity would be of interest.

Further, our analyses of data from SourceForge[3] indicate that language fragmentation may negatively affect developer productivity by as much as forty or fifty percent [56]. However, considering the fact that productivity has been shown to depend on an exceptionally large number of factors [46, 72], a fifty percent dependence on any one factor is unlikely. Nevertheless, these early results highlight a rich area of investigation, which this research pursues.

Fortunately, as we strip away the effects of confounding factors, we continue to find a practically significant correlation between language fragmentation and developer productiv-

[2] *Companion languages* (as we define the term) are programming languages which are commonly used together to create a particular type of software solution. Each language in a companion set meets specific needs within the solution type. For example, Java web applications (a solution type) often utilize SQL for database communication, JavaScript for client-side processing, and HTML/CSS for generating the application interface.

[3] SourceForge is a large open source software community from which we are able to obtain source code and revision histories from thousands of projects—including for example PDFCreator, BitTorrent, and FileZilla.

ity. Ultimately, this research seeks to fully isolate the fragmentation-productivity relationship in order to determine causality. If a causal relationship is found, then language fragmentation would need to be tested against other productivity metrics in order to be incorporated into existing productivity and cost-estimation models [15].

1.3.2 Industry Best Practices

Analyzing data from industry projects would allow us to ask a number of important questions, including:

- If my company is maintaining a large code base in programming language X, how will my developers' productivity be affected by an additional project in language Y?
- 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?

Development directors and programming team managers 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. Ideally, an exploration of industrial projects would lead to the formulation of new Software Engineering best practices.

1.4 Related Work

In general, we have not found any other research that attempts to measure programming language fragmentation or to examine its effects on developer performance. However, there are a number of subtopics within this thesis that rely on previous research.

1.4.1 Effects of Natural Languages on Problem Solving

In the introduction we identify two linguistic studies [6, 10] that report finding connections between natural language and human cognition. In particular, both of these studies find that natural language affects problem solving in humans. One of these studies [10] also reports that balanced bilinguals possess a greater degree of attention control, which is a necessary component of problem solving. Although the idea that language impacts cognition seems generally accepted within the field of linguistics, studies that focus on the problem-solving abilities of mono-, bi-, and multilingual speakers are scarce in the literature. Further, the exact relationship between natural languages and programming languages is unclear. Although some researchers lump the two classes together [27], others contend that programming languages are not actually languages in the same sense as are natural languages [70, 71]. Consequently, we must be cautious when applying linguistic research to the study of programming languages. Within this thesis, we use linguistic studies primarily to motivate the research topic.

1.4.2 Using Entropy to Measure Spread

In order to empirically evaluate the relationship between language fragmentation and programmer productivity, we require a metric that effectively characterizes the distribution of a developer's efforts across multiple programming languages. To meet this requirement, in Chapter 2 we develop an entropy-based metric, *language entropy*, for measuring language fragmentation. Our metric is based primarily on a metric which Taylor, Stevenson, Delorey, and Knutson [89] formulated to characterize the spread of author contributions within program files and across software projects. As a measure of disorder in a system, the concept of entropy is sometimes used in Software Engineering to represent spread, or the evenness with which elements are distributed across a system. The entropy model employed by Taylor et al. was originally introduced into Software Engineering from information theory [40], which defined entropy in terms of probability theory. Studies utilizing probability-based entropy metrics appear sporadically in the Software Engineering literature [11, 41], but none of them apply entropy to language fragmentation.

1.4.3 Productivity Metrics

For more than forty years researchers have formulated metrics and models for measuring programmer productivity. Numerous studies have shown that productivity is impacted by tens, or even hundreds of factors [46, 72]. Thus, no one model is sufficient. In fact, all reasonable productivity metrics correlate to some degree, and all metrics suffer from threats to validity. The works of Conte, Dunsmore, and Shen [22] and Endres and Rombach [36] discuss the properties of common productivity metrics and overview their limitations. This thesis employs lines-of-code (LOC) and time primitives to measure productivity (see Chapter 3). These primitives are popular in the literature [14, 21, 33, 91] for several reasons, including: 1) the availability and accessibility of relevant data, and 2) in most contexts these primitives correlate well with other productivity metrics.

1.4.4 Data Sources

The data set used in this thesis was previously collected from the August 2006 SourceForge Research Archive (SFRA). For details on the data source, summary statistics, collection tools/processes, and other pertinent and interesting analyses of the data, see papers by Delorey et al. [26, 28, 29].

1.5 Thesis Statement

In this thesis 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 individual programmer productivity within the SourceForge community—thereby demonstrating that to maximize individual programmer productivity, language-fragmented projects should minimize the number of languages that each developer must use by either reducing fragmentation within the project, or by assigning developers to specific languages.

1.6 Project Description

We address the relationship between language fragmentation and programmer productivity within two different contexts: 1) the individual developer, and 2) the overall project. The first context represents a direct approach to answering the question of interest. The second is intended to provide validation of the first. However, as we show in Chapters 4 and 5, specific complications require future work before we can form conclusions within the project context. Accordingly, we spend much of the latter half of this thesis uncovering and defining these complications, as well as outlining potential solutions.

Further, since project productivity is generally considered more important than individual developer productivity (from an industry perspective), and considering that each context is subject to different threats to validity, replicating the study in both contexts should allow us to form more robust conclusions. In particular, consistent results between the individual and project contexts would represent stronger evidence than that obtainable by studying programmers in isolation. However, *in*consistent results would also be valuable. For instance, to find a negative effect within the programmer context and a positive effect within the project context would suggest that the overall project-level benefits of language specialization outweigh the negative impact of language fragmentation on developers. Such results would prompt further study of the data to determine whether assigning developers to specific languages within a project would increase the positive impact of language specialization on project productivity.

In order to empirically evaluate the relationship between language fragmentation and programmer/project productivity within the SourceForge community, we:

1. Develop metrics for measuring productivity and language fragmentation.
2. Select data suitable for calculating the needed metrics.

3. Develop and validate statistical models that isolate the correlation between language fragmentation and *individual* programmer productivity.
4. Develop additional methods to mitigate threats to validity within the developer context.
5. Explore limitations that need to be addressed in future work for effective analysis of language fragmentation within the project context using the SourceForge data set.

1.6.1 Metrics

We use a standard Software Engineering approach to estimating both programmer and project productivity—LOC per unit time (see Chapter 3). To measure language fragmentation, we adapt a probability-based entropy metric from information theory, based on lines of code, which we refer to as *language entropy* (see Chapters 2 and 3). Since entropy is designed to measure the relative spread of elements across a set of classes within a system, it is well-suited to modelling the spread of a developer's efforts across multiple programming languages. Conceptually, within the individual developer context, the system is the total source code contribution during a specific month, the classes are the programming languages used by the developer, and the elements are the individual lines of code from the total contribution. Similarly, within the project context, the system is the total of all source code contributions made to a project during a specific month, the classes are the programming languages used during that month, and the elements are the individual lines of code.

1.6.2 Data

This thesis is a first step toward understanding the effects that language fragmentation has on programmer performance. As such, one of its primary purposes is to justify and motivate deeper research. Thus, despite the limitations of using open source software data—e.g., the researcher is unable to interview developers, as would be possible in a company setting—the accessibility and low-cost of such data make it an excellent starting point. For this project we utilize data from the SourceForge Research Archive (SFRA). The data set is large enough to obtain precise statistical results and provides sufficient detail to calculate both language entropy and programmer productivity.[4]

1.6.3 Statistical Models

Statistically analyzing the relationship between language fragmentation and productivity requires selecting appropriate statistical models and adapting them to the topography of the data. Because statistical models are generalizations of data—intended to represent major features, while abstracting away noise—the general shape of the data must be taken into account. In some cases the data require transformation before they are suitable for analysis. We also identify and address outliers and adjust for serial correlation because time is a factor in our metrics. In short, the data require advanced statistical techniques, for which there can be disagreement, even among experts, regarding the best approach. Consequently, we collaborate with the Brigham Young University Department of Statistics in preparing our analysis, and we make every effort to clearly explain our selection of statistical tools.

In addition to developing the statistical models, we also validate in Chapter 3 that the models accurately measure the relationship of interest. For instance, one concern is that the data violate a key assumption of regression tools, which require a uniform range in the data.[5] Therefore, we test the effects of model assumption violations to estimate their impact on the results.

1.6.4 Validation

The thesis incorporates two forms of validation: 1) statistical, and 2) contextual.

[4]The data set contains 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.

[5]The language entropy and LOC-based productivity metrics actually map the data into a non-uniform, discontinuous space.

Statistical Validation

Statistical validation rests on two premises: 1) the statistical analysis efficiently and accurately models the relationship of interest, and 2) the mathematical complexity of the entropy metric does not confound the statistical analysis. The first premise is easily tested using p-values and confidence intervals, which are incorporated into standard statistical methodology. To test the second premise we generate artificial entropy data, bearing the same characteristics and dimensions as the true data, but for which we already know the relationship between language fragmentation and developer productivity (i.e., zero-correlation data, based on a random selection of values). By applying our statistical models to the artificial data, we are able to separate the effects of the entropy metric on model parameters from the effects of the actual relationship of interest on those parameters. Understanding the interaction between the statistical tools and the mathematically complex entropy metric allows us to compensate for adverse effects (see Chapter 3).

Contextual Validation

Although standard statistical methods incorporate validation techniques, statistics cannot account for missing information. When possible, researchers test hypotheses using controlled, randomized experiments to account for unmeasurable and/or uncontrollable variables. In software engineering, however, randomized experiments are often prohibitively expensive and, therefore, must be justified by preliminary observational studies.

As an observational study, this thesis suffers from the exclusion of potentially important factors in the statistical model. To help account for missing and unmeasurable variables that may contribute to the relationship between language fragmentation and developer performance, we address the question of interest from two different contexts: 1) the individual developer, and 2) the overall project. We explore the programmer context in Chapters 2 and 3, and in Chapters 4 and 5 we develop methods for appropriately analyzing language fragmentation within the project context. From the individual programmer context we address the question of interest directly—from the project context, indirectly. Although each context suffers from excluded variables, together they should help to validate each other—for example, many of the programmer context unknowns are mitigated in the project context.

This method of dealing with threats to validity through multiple replications—i.e., "families of experiments" [9]—is an appeal to parsimony and "is based on the assumption that the evidence for an experimental effect is more credible if that effect can be observed in numerous and independent experiments each with different threats to validity" [9, p. 472] [20]. We discuss the replication methodology in detail in Chapter 6.

1.7 Overview of Findings

The original analysis of language fragmentation with respect to individual programmer productivity has been strictly replicated twice during the course of this thesis, each time using a new random sample from the SourceForge data set. The original analysis was conducted as part of our efforts to develop a metric for measuring language fragmentation, the results of which are not included in this thesis. The second iteration (i.e., the first replication; Chapter 2) corrects for several analysis errors and utilizes more advanced statistical techniques. The third iteration (Chapter 3) adds much-needed validation procedures, including controls for additional confounding factors, and extensively explores the relationship between language fragmentation and programmer productivity. In general, the two replications refine the original analysis procedures, introducing successively deeper explorations of contextual variables (e.g., alternative programmer productivity metrics) and catching errors (such as biased data sampling in the original experiment).

All three iterations of the study find statistically and practically significant *negative* correlations between language fragmentation and individual programmer productivity—although the correlation magnitudes decrease slightly with each replication, due to additional corrections for confounding factors. As of the third iteration, we find that *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).*

Chapters 4 and 5 primarily address the fragmentation-productivity question within the project context. However, due to limitations of the data set, which require development of new analysis techniques (particularly filtering techniques), these chapters are spent exploring problems and outlining solutions; the actual analysis of language fragmentation with respect to project productivity is left for future work. Chapters 4 and 5 also further attack the validity of the findings for the programmer context, outlining additional concerns and potential analytical adjustments; responses to these criticisms are also left for future work.

The last chapter addresses replication as a guiding research methodology. The topic of replication is a controversial issue for the Software Engineering research community. Consequently, the final chapter of this thesis is the start of a larger work on replication as an empirical methodology, including adaptations of that methodology specific to Software Engineering research. Chapter 6 argues in favor of replication, claiming that all studies of practical interest should be replicated and that good science is always founded on replication principles, whether recognized or not. As far as language fragmentation is concerned, we are still in the process of discovering that which "we don't know we don't know" [4, p. 19, paraphrased]—i.e., which variables affect our study conditions and how those variables potentially confound the results. This discovery process is defined by replication.

1.8 Publications

Table 1.1 lists the five publications resulting from this research along with corresponding thesis chapters. Note that because the chapters of this thesis are published as a series of independent papers, some background information is, by necessity, repeated in multiple papers. Additionally, though the content remains unchanged, the texts of Chapters 4 and 5 have been updated since publication for inclusion in this thesis.

Chapter 2	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. In *4th International Workshop on Public Data about Software Development*, June 2009. [56]
Chapter 3	Jonathan L. Krein, Alexander C. MacLean, Charles D. Knutson, Daniel P. Delorey, and Dennis L. Eggett. Impact of programming language fragmentation on developer productivity: A SourceForge empirical study. *International Journal of Open Source Software and Processes*, 2(2):41–61, 2010. [57]
Chapter 4	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 *1st International Workshop on Replication in Empirical Software Engineering Research*, May 2010. [62]
Chapter 5	Alexander C. MacLean, Landon J. Pratt, Jonathan L. Krein, and Charles D. Knutson. Trends that affect temporal analysis using SourceForge data. In *5th International Workshop on Public Data about Software Development*, June 2010. [63]
Chapter 6	Jonathan L. Krein and Charles D. Knutson. A case for replication: Synthesizing research methodologies in software engineering. In *1st International Workshop on Replication in Empirical Software Engineering Research*, May 2010. [55]

Table 1.1: Publications resulting from this research (with corresponding thesis chapters).

1.9 Future Work

The following notable items could not be appropriately addressed elsewhere in this thesis and hence are listed here as suggestions for future work. (This list is not comprehensive; see the individual papers for additional ideas.)

1.9.1 Marginally Active Developers

Based on research by Healy and Schussman [42], as well as on our own analysis of the SourceForge data, we believe that marginally active developers may significantly affect the analysis results. Healy and Schussman report "spectacularly skewed" distributions for numerous project activity metrics (e.g., number of revisions per project). Our own investigations indicate that approximately one-third of the developers contribute eighty percent of the data. Large numbers of marginally active developers can pose several problems: First, we believe that marginally active developers are less likely to write in multiple languages during a given month. Consequently, the results of a comparison between single- and multi-language contributions would be misleading. Second, due to a lack of data, the estimated contribution averages for the marginally active developers are much more likely to suffer from sampling error. Third, the marginally active developer population is far less interesting to study from a productivity standpoint than full-time, professional developers. Consequently, a differentiated replication with only "active" developers may be particularly enlightening.

1.9.2 Entropy Metric Complexity

The entropy calculation appears ideal for measuring language fragmentation. As a diversity metric, it captures both the *richness* of entity types (i.e., the number of programming languages used by a developer) and the *evenness* with which the entities in a set spread across those types (i.e., how evenly developers spend time programming in each language). However, the metric's use of logarithms adds significant complexity to the overall analysis, and since misinterpreting the metric would mean misinterpreting the results, we recommend further validating the metric. One approach would be to substitute a simpler, more coarse-grained metric for the complex (presumably more precise) entropy metric and test for corroborating results. We propose *language count* (a strict measure of richness) as a candidate surrogate with which to validate the language entropy metric.

1.9.3 Omitted Data

In this thesis we compute productivity and entropy scores using the *lines_added* statistic, as reported by CVS logs (the source code revision control system used by SourceForge at the time our data were gathered). However, as discussed in Chapter 4, we have since discovered that the *lines_added* statistic only records changes to files that have been previously checked in to CVS. Lines added as a result of new file check-ins are recorded exclusively by the *initial_size* statistic. Therefore, our analysis neglects a portion of the available data (new file commits), which likely biases the results toward software maintenance activities. The full implications of this omission have yet to be explored. Future work may find value in replicating the analysis with the full data, as well as with only the initial file check-in sizes.

1.9.4 Variance in Programming Language Verbosity

Programming language verbosity varies based on syntax, built-in features, standard practices, and the use of libraries (e.g., Perl regular expression libraries versus manually parsing C-strings). In fact, Delorey, Knutson, and Giraud-Carrier [28] found that the three most common two-language profiles in SourceForge are C/Perl, C/C++, and JavaScript/PHP. The concern is that when a developer writes in two (or more) languages, if the secondary language is particularly terse (or conversely verbose) relative to the primary, then the developer's productivity may appear to decline (or conversely increase), when in fact the logical productivity has not changed significantly. For example, suppose a developer writes predominantly in C, but breaks out occasionally into Perl. During those months that s/he writes in Perl, LOC production may decrease simply because s/he is spending time writing in a terse language. If this example holds true for many developers, then our results may be biased. For consideration, we suggest that average commit size per language might make a reasonable normalization factor. For three highly-related discussions of the effects of programming language verbosity on LOC productivity, we also recommend the works of Brooks [19], Jones [45], and Delorey, Knutson, and Chun [26].

1.9.5 Inference Limitations

Our inferences are limited to developers on SourceForge, and consequently, conclusions about other software development environments (e.g., in the corporate domain) must be treated skeptically. Additionally, as an observational study, the project does not support causality inferences, though the findings are suggestive. To further generalize these conclusions, future projects should replicate the methods of this thesis in other development environments (including corporate settings) and/or run controlled, randomized experiments.

1.9.6 Other Productivity Metrics

Despite its general utility in Software Engineering, the LOC/month metric is only one of many programmer productivity metrics. Since all metrics suffer limitations, a future project may build on this work by extending the analysis to other productivity models (see Chapter 3).

1.9.7 Taxonomy of the SourceForge Community

Because Open Source Software (OSS) data are publicly available—including source code revision histories, bug reports, email lists, etc.—much of the current research in Software Engineering relies on repositories like SourceForge. In fact, the SourceForge Research Data Archive (SRDA) alone boasts more than one hundred researchers [3]. Nevertheless, most research studies utilizing OSS data are forced to treat their subject pool, more or less, as a single population, simply because no one has yet classified the sub-populations. Considering that SourceForge hosts more than a hundred thousand projects [3], treating it as a single population is probably a gross oversimplification[6]—one that quite possibly contributes to the still pervasive problem in Software Engineering of findings that do not generalize well [53, 54, 55, 47, 86].

In this regard, our analysis of language fragmentation using SourceForge data indicates that authors contribute almost exclusively to projects of a similar size (referred to as *author project size bridging*; Chapter 4), indicating that project size might make a good variable on which to pivot a taxonomy of SourceForge projects. More generally, SourceForge data affords numerous variables, as well as qualitative data, from which to create a taxonomy

[6]For example, consider the writings of Raymond [78] versus recent work by Bird, Pattison, D'Souza, Filkov, and Devanbu [13].

of the SourceForge community; such a taxonomy would lend significant insight into previous and future OSS studies.

1.9.8 The Project Context

In this thesis we were not able to complete an analysis of language fragmentation within the project context. In particular, the issue of "cliff walls" [62, 63] threatens all studies utilizing SourceForge data to examine evolutionary project growth. As we discuss in Chapters 4 and 5, there are numerous causes of cliff walls in the data. In some cases whole projects need to be excluded from analysis, in other cases specific commits should be excluded, and sometimes large commits may be interpretable by combining data from multiple project branches. In this thesis we have laid out what we believe to be the primary causes of cliff walls, background knowledge on those causes, and potential solutions. On that foundation, future work may develop new methods for large-scale evolutionary analysis of SourceForge project data.

Chapter 2

Language Entropy: A Metric for Characterization of Author Programming Language Distribution[1]

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 find 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 Introduction

The ultimate deliverable for a software project is a source code artifact that enables computers to meet real-world needs by solving a set of complex problems. The process of software development, therefore, involves both problem solving and communication of solutions to

[1]This chapter is published in the proceedings of the *4th International Workshop on Public Data about Software Development*, 2009 [56]. Because each chapter is published as a separate paper, many of the motivational and background concepts are repeated. Consequently, Sections 2.1 and 2.2 of this chapter may be skimmed. Also, most of the future work at the end of this chapter is summarized in Chapter 1. The purpose of this chapter is to establish the entropy metric for characterizing language fragmentation. Hence, the analysis presented in this chapter is preliminary, and the relationship between language fragmentation and programmer productivity is more thoroughly explored in Chapter 3.

a computer. We believe that the languages by which humans communicate solutions to computers may in fact play a role in the complex processes by which they generate those 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, pp. 240, 249]. Although their study focused on the effects of *natural* language on problem solving, their concept of language is highly representative of those used in programming activities. Other research in the area of linguistics examines the differences between mono-, bi-, and multilingual speakers. One particular study [10], focusing on the differences between mono- and bilingual children, found some specific differences in the subjects' abilities to solve problems.

These linguistic studies prompt us to ask questions about the effects of working concurrently in multiple programming languages on the problem-solving abilities of developers. However, to begin studying that relationship, we first develop a metric that characterizes the distribution of an author's development efforts across multiple programming languages.

In this paper, we present *language entropy* as a candidate metric for measuring the distribution of languages within an author's programming contributions. We further provide preliminary support for a relationship between this metric and the problem-solving abilities of developers by demonstrating a significant relationship between language entropy and programmer productivity, as measured in lines of source code added to projects per month.

2.2 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.

2.3 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 work by Taylor, Stevenson, Delorey, and Knutson [89].

2.3.1 Definition

Entropy is a measure of chaos in a system. The concept of entropy originated in thermodynamics but has been adopted by information theory [40]. 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.

2.3.2 Calculation

The general form for entropy is shown in Equation 2.1.

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

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 2.1, produce a parabolic curve. Three languages produce a three-dimensional shape. Entropy calculations beyond three dimensions are difficult to visualize.

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 2.1[2]. We then calculate the result of Equation 2.1 using those language proportions.

2.3.3 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

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

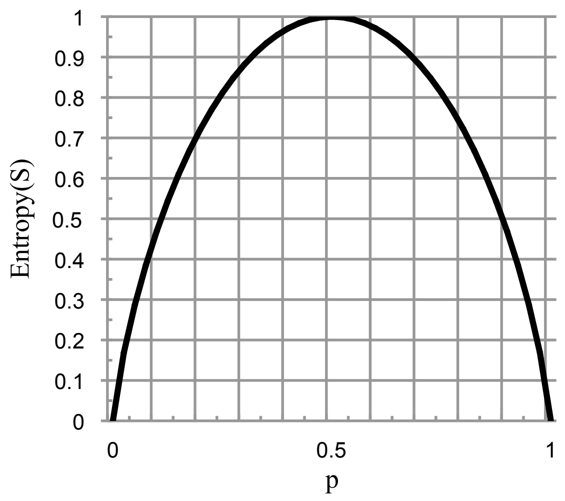


Figure 2.1: 2nd order entropy curve.

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 2.2, 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.[3] However, the effect of adding an additional language diminishes as the total number of languages increases (see Equation 2.3 and Table 2.2).

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

Conversely, the minimum possible language entropy is always 0, indicating that the author only added lines of code in a single language (see Table 2.2).

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

# 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.

2.4 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 work by Delorey, Knutson, and MacLean [29].

2.4.1 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 [29].

A study by Delorey, Knutson, and Chun [26] 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 2.3 shows the 10 most popular languages with rankings. Delorey et

	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.

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.

2.4.2 Producing a Data Sample

From the initial data set we extracted a random sample of 500 developers[4] 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

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

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 ten factors identified by Delorey et al. [26], 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.

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 Delorey et al. [26] 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.

2.5 Analysis

2.5.1 Transforming the Data

Figure 2.2 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, which depicts the transformed data, that there are only minimal outliers, the range is controlled, and the distribution is approximately normal.[5]

[5]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.

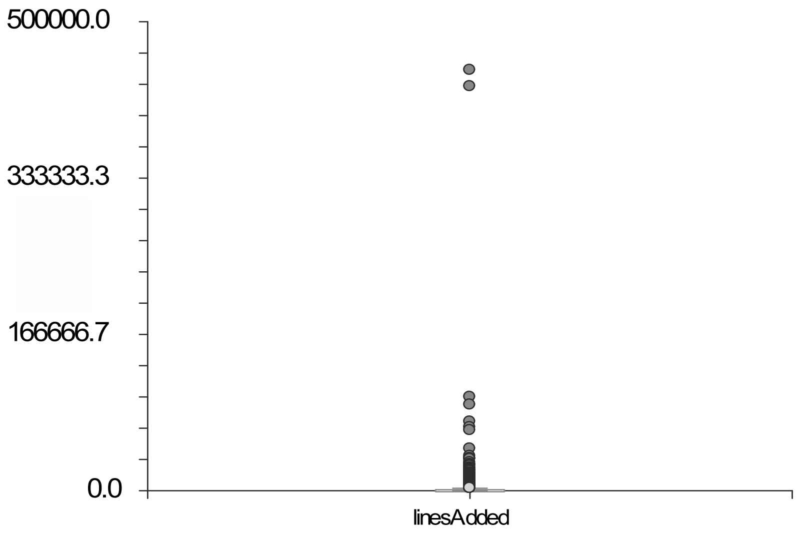


Figure 2.2: Box plot of lines added.

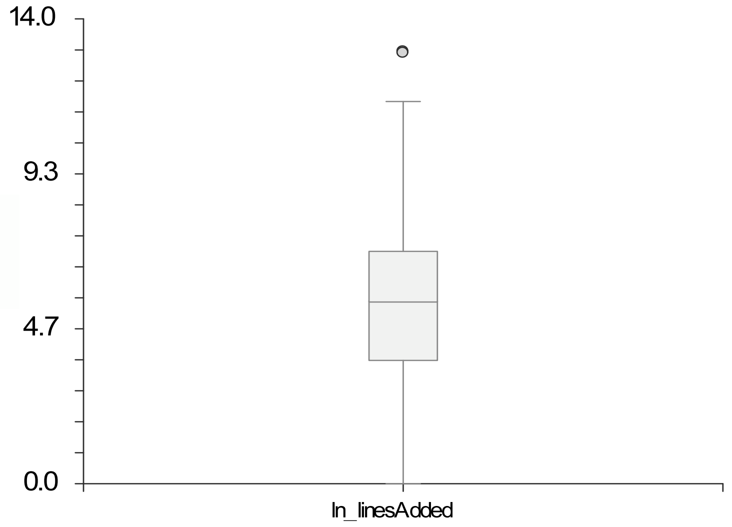


Figure 2.3: Box plot of ln(lines added).

2.5.2 Selecting a Statistical Model

Figure 2.4 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 2.4) is masked by crowding, which causes points to be plotted over other points. In total, there are 3,940 points plotted,

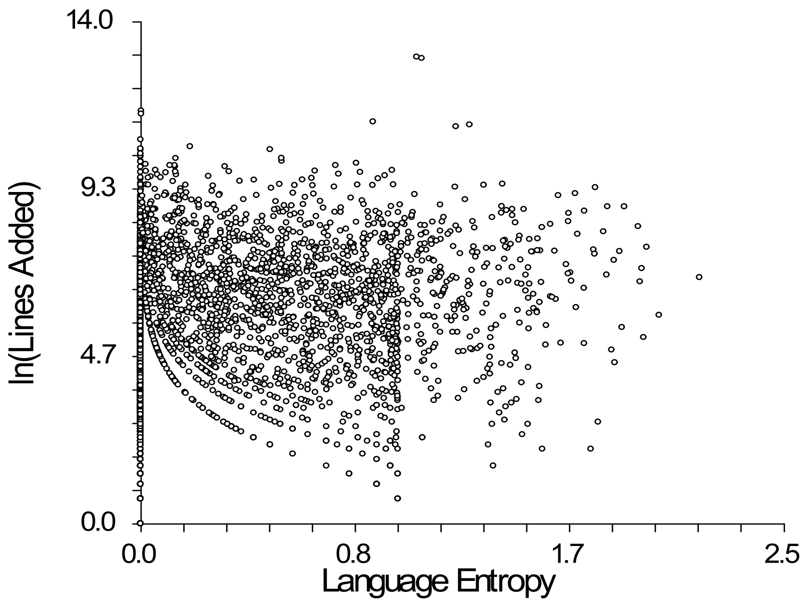


Figure 2.4: Plot of ln(lines added) vs. language entropy.

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.[6]

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

[6]The distribution of the data points with respect to language entropy is fairly consistent with the findings of Delorey, Knutson, and Giraud-Carrier [28], 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.

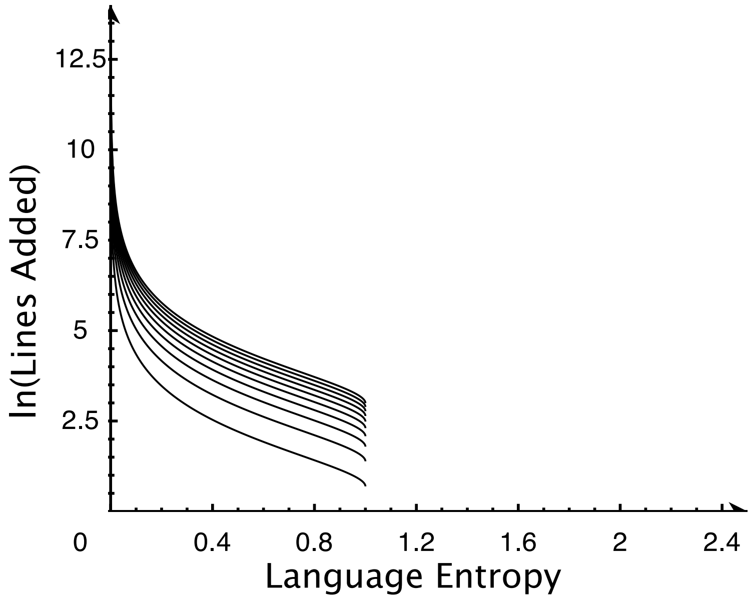


Figure 2.5: Graph of the first 10 entropy curves for the 2 language case.

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 2.5 shows a graph of the first 10 equivalence classes.

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.

2.5.3 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.

2.6 Results

Table 2.5 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 2.4.1), 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.

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.

In Table 2.5, 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 2.7.4 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.

2.7 Limitations

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

2.7.1 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 2.4.2). 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.

2.7.2 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).

2.7.3 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.

2.7.4 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.

2.8 Future Work

In this section we outline avenues for future research.

2.8.1 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 2.8.2), will provide meaningful results from which practitioners may make better-informed decisions regarding project-developer assignments and the adoption of new languages and frameworks.

2.8.2 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?

2.8.3 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.

2.8.4 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?

2.8.5 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 [15].

2.9 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[1]

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

[1]This chapter is published in the *International Journal of Open Source Software and Processes*, 2010 [57]. The focus of this chapter is on utilizing the entropy metric to assess the relationship between language fragmentation and programmer productivity, whereas the focus of Chapter 2 was on developing that metric. Although significantly improved, much of the setup and protocol in this chapter is similar to that in Chapter 2 (repeated for publication). Sections 3.1, 3.3 through 3.5, and the beginning of 3.6 may be skimmed, as well as the limitations and the future work sections at the end. Sections 3.2 and 3.6.4 are completely new. Pay particular attention to Section 3.6.4, which presents a detailed validation of the entropy metric with respect to the statistical model; that validation includes an important caveat for interpreting the results. Results are based on a new data sample from the SourceForge data set.

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, pp. 240, 249]. 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 [10], focusing on the differences between mono- and bilingual children, found specific differences in the subjects' abilities to solve problems. 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" [44, p. 12]. 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 [72] performed one of the earliest studies to identify programmer productivity factors. Nelson found that programmer productivity correlates with at least 15 factors. More recently (2000), Capers Jones [46] identified approximately 250 factors that he claims influence programmer performance. Summarizing this research, Endres and Rombach state that reducing productivity to "ten or 15 parameters is certainly a major simplification" [36, p. 190]. 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 correlate 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 [22] and to work by Endres and Rombach [36]. 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 [14, 21, 33, 91] 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 [56]. 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 [89].

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 [40, 81]. 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 [11, 41], 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).

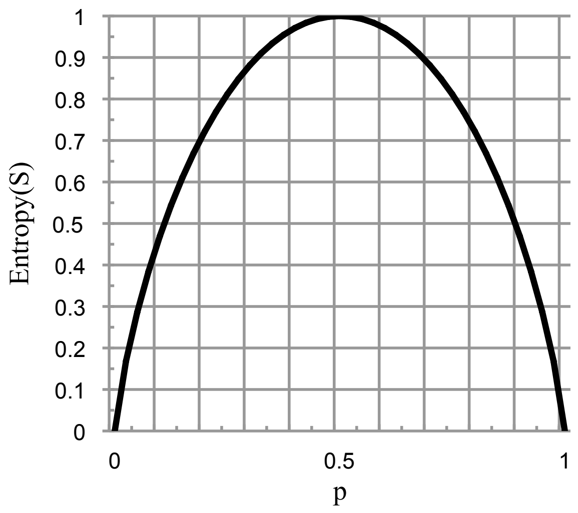


Figure 3.1: Relationship between entropy and language proportion for two languages.

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 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

# 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.

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[2] zero, indicating that the developer contributed in only one language (see Table 3.1).

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.

[2]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.

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.
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 [29].

	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).

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 [29].

A study by Delorey, Knutson, and Chun [26] 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; 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. [26]—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. [26], 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.

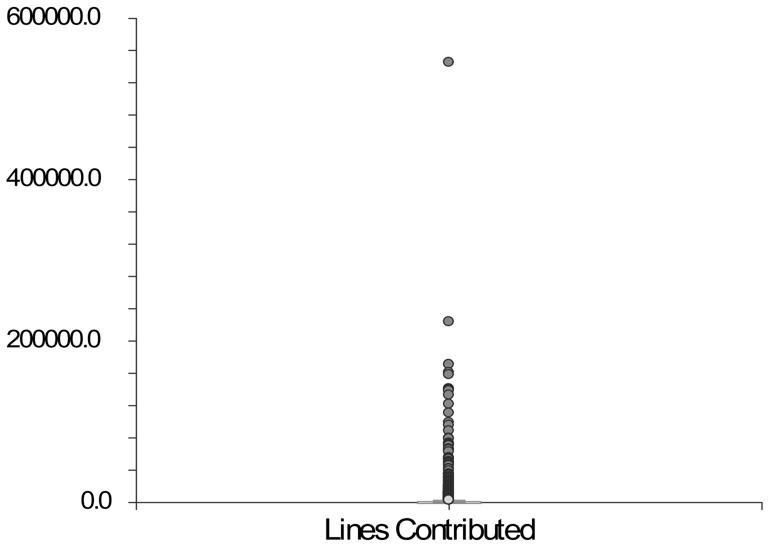
3.6 Analysis

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.

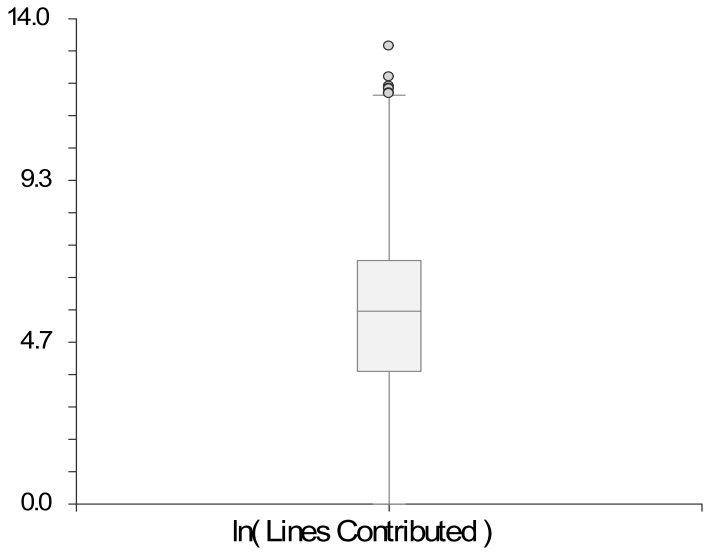
3.6.1 Transforming the Data

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.[3]

[3]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.



(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.

3.6.2 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 [28], who report for the same SourceForge data set that approximately 70% of developers write in a single language per year.

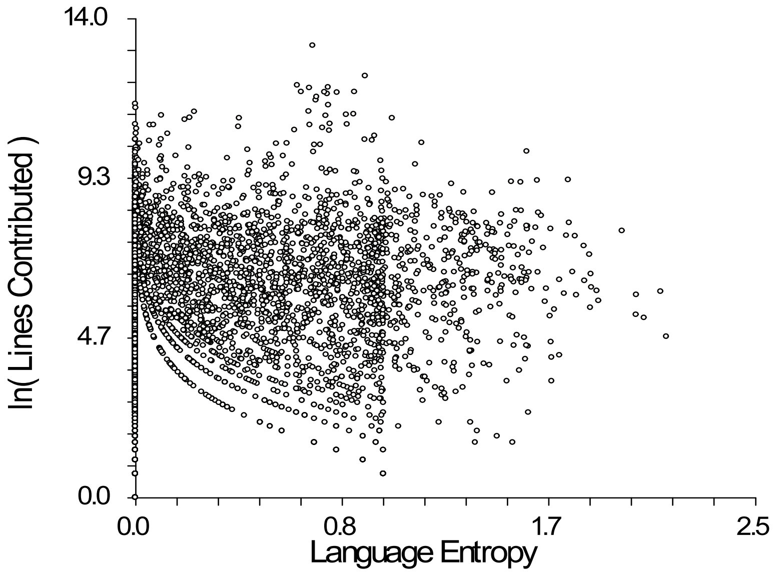


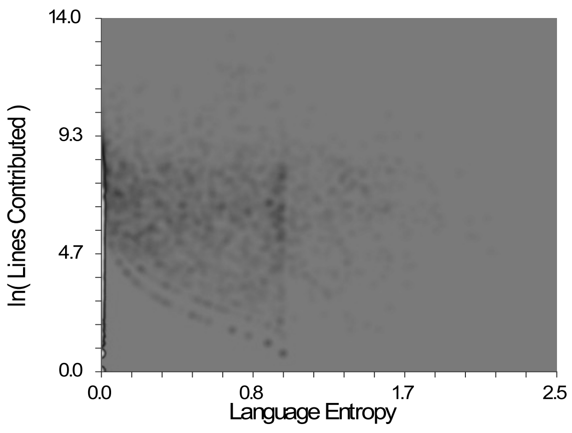
Figure 3.3: Plot of ln(lines contributed) vs. language entropy.

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

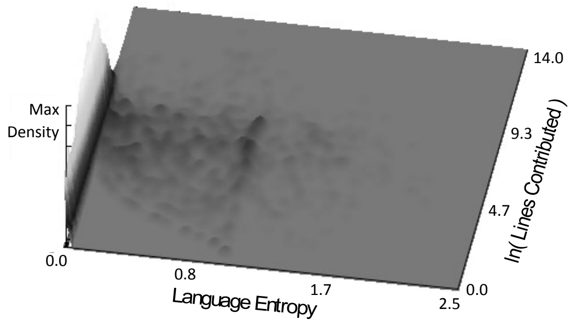
Table 3.3: Distribution of data points by language entropy.

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.

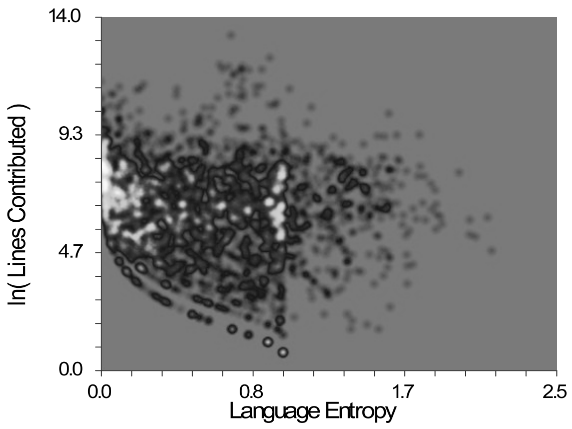
Since this study intends to show a significant relationship between language entropy and lines contributed, we must demonstrate both a significant correlation between the two metrics and a reasonable variance in the data. The data plot and density maps, however, show a large spread in the data, indicating considerable variance. For the non-zero-entropy



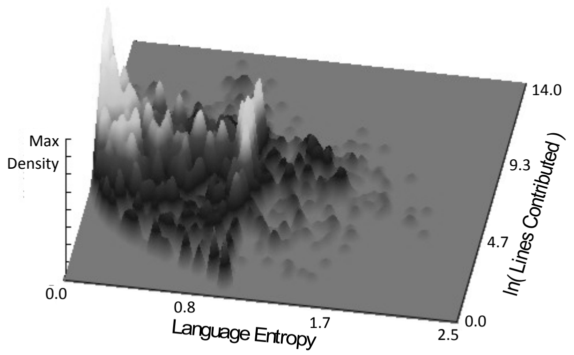
(a) Density *heat* map; data at $E(S) = 0$ dominates.



(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.

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 [76, 23, 30, 65]. 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.

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 [77]. However, to be conservative we assume that serial correlation exists in the data and adjust for it in our analysis.

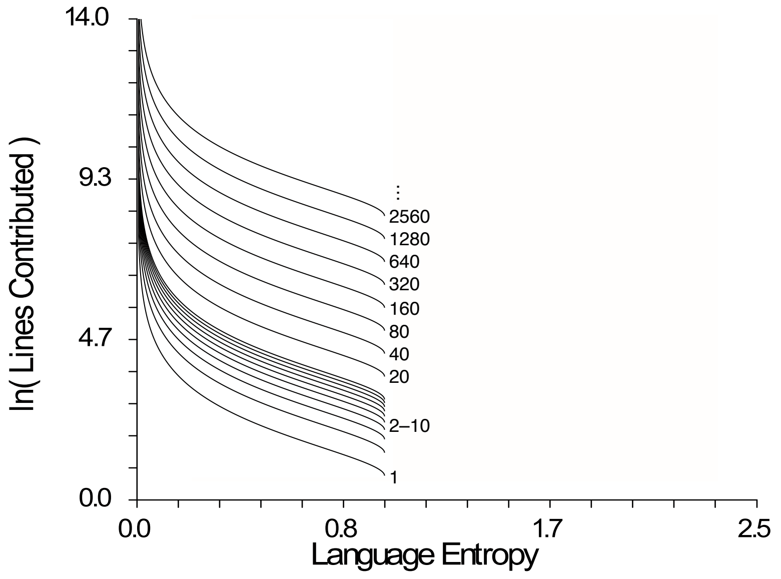
3.6.4 Banding in the Data

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).

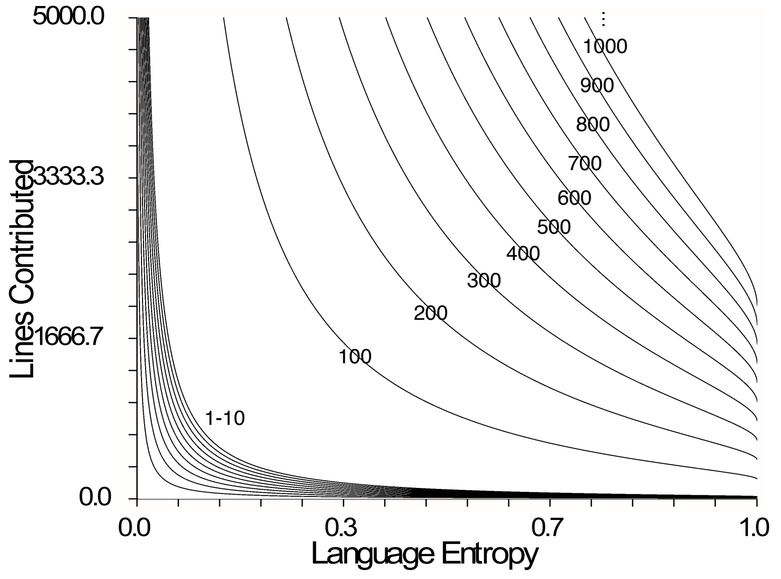
Figure 3.5(a) also demonstrates that as the equivalence classes progress in the positive y-direction they grow exponentially closer together. By the fourth equivalence class the bands visibly blend on the graph. Thus, even though the bands are discrete in the y-direction, the space between them quickly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

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.



(a) Log scale



(b) Normal scale

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

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 (expected 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 et al. [28] found in their analysis of the SourceForge data set that only 10% of developers use more than two languages per year. 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).

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.

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, *nonZeroEntropy-*

GroupSlope, 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.

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

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

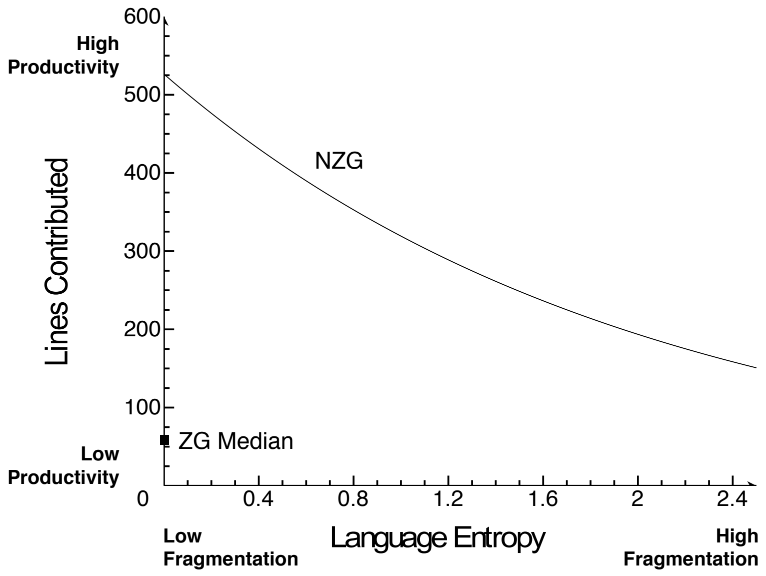


Figure 3.6: Best-fit model on the normal scale.

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, 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 [42] in SourceForge data). 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?[4]
- 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?

[4]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 [15].

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[1]

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 effects that they have on the results of our previous paper entitled "Impact of Programming Language Fragmentation on Developer Productivity: a SourceForge Empirical Study." As a result, we question the validity of all studies utilizing SourceForge data for evolutionary analysis of project growth.

4.1 Introduction

The present work began as a replication of a study by Krein, MacLean, Delorey, Knutson, and Eggett [57], "Impact of Programming Language Fragmentation on Developer Productivity: a SourceForge Empirical Study," in which the authors explore a potential relationship between individual developer productivity and the 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. Though language fragmentation appears to negatively impact individual developers, it may, for instance, positively impact

[1]This chapter is published in the proceedings of the *1st International Workshop on Replication in Empirical Software Engineering Research*, 2010 [62]. Although the content remains unchanged, the text of this chapter has been updated since publication for inclusion in this thesis.

overall projects. Such a replication study would obviously shed light on the potential impact of writing software in only one language, as opposed to programming 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 requires 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 discuss limitations of SourceForge data that must be considered before exploring language fragmentation with respect to project evolution. We also outline potential threats to the validity of our original study of language fragmentation's impact on individual developer productivity. Thus we discuss the issue of language fragmentation both from the context of the individual developer (as in the original study) and from the context of the overall project. Though both contexts suffer from limitations in the data set, we contend that the limitations are more severe for questions relating to project evolution.

4.1.1 SourceForge as a Data Source

More than 100 researchers use the SourceForge Research Data Archive (SRDA) [3] hosted at Notre Dame to analyze development behavior in open source projects. Howison and Crowston [43] enumerate several pitfalls in using SourceForge as a data source for research. In Section 4.2 we extend their analysis with insights into the limitations of SourceForge for studying specific project attributes.

4.1.2 Data Set

For this study, as well as the original [57], the data set (which we refer to as the *sample*) comprises all projects designated "Production/Stable" or "Maintenance" as of the October 2006 SRDA dump of SourceForge. Thus development histories are included for all projects from project inception through October 2006. The data was originally collected and preprocessed in a related work by Delorey, Knutson, and MacLean [29].

4.1.3 Language Fragmentation and Productivity Metrics

The targeted study [57] explores the correlation between language fragmentation and programmer productivity. Fragmentation is measured by language entropy (originally defined by Krein, MacLean, Delorey, Knutson, and Eggett [56]), with productivity defined as the number of lines of code committed to all "Production/Stable" or "Maintenance" phase projects on SourceForge in a given month.

4.1.4 Language Entropy

Entropy, defined in Equation 4.1, measures the "evenness" and "richness" of a distribution (p) of classes (c) in a system (S). For *language* entropy, "evenness" describes how evenly a developer contributes code across one or more languages. For example, if a developer writes 100 lines of Python and 100 lines of Java in one month, s/he would have the maximum possible entropy for two languages: 1.0. For 150 lines of Python and 50 lines of Java, the language entropy would be 0.811. "Richness" describes the number of languages used by an author in a given month. Maximum possible entropy is a strict function of richness, as defined in Equation 4.2, where c is the number of languages (classes).

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

$$E(S)_{max} = \log_2 c \tag{4.2}$$

The original language fragmentation study reports a negative correlation between language entropy and programmer productivity.

4.1.5 Definitions

In this study we define two terms precisely.

Definition 1. Daily Commit *is a unit of work—the net contribution 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 find that daily commit unmasks certain attributes of the data that are not visible at coarser granularity.

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

4.2 Project Attribute Pitfalls

Two attributes of SourceForge projects potentially lead to biased results if not adequately controlled: 1) unusual project growth patterns (which we refer to as *cliff walls*), resulting from several routine development conditions including auto-generation of source code, internal development, and development pushes; and 2) related to the first issue, small projects. To address the first concern, we present the Java eXPerience FrameWork (JXPFW) project as an illustrative example of how cliff walls in SourceForge data can lead to biased results for language fragmentation studies specifically, and for project evolution studies generally. After articulating problems in the example case, we show that the problems exist in a large percentage of projects on SourceForge. Finally, we address the issue of small projects and their relationship to cliff walls.

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 has the largest daily commit.

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

4.2.2 Cliff Walls

Abnormal growth spikes in a project (which we refer to as *cliff walls*) constitute a serious threat to validity in evolutionary research utilizing SourceForge project data. These growth spikes represent periods of time during which data for a project is masked, missing, or ambiguous. Figure 4.1 shows the growth of the Java eXPerience FrameWork over time for *all* lines of "code," including non-source and auto-generated lines. The figure indicates that although the project was created on September 8, 1999, more than two and a half years passed before any code was committed. Figure 4.2 shows the growth of JXPFW over time with non-source and auto-generated lines excluded, indicating that on May 1, 2002, 138 new *source* code files were added to the project by a single author, totaling 18,675 lines of code. As we discuss in the following sections, auto-generated source code, internal development, and development pushes are all developmental practices that lead to abnormally large commit sizes (i.e., cliff walls).

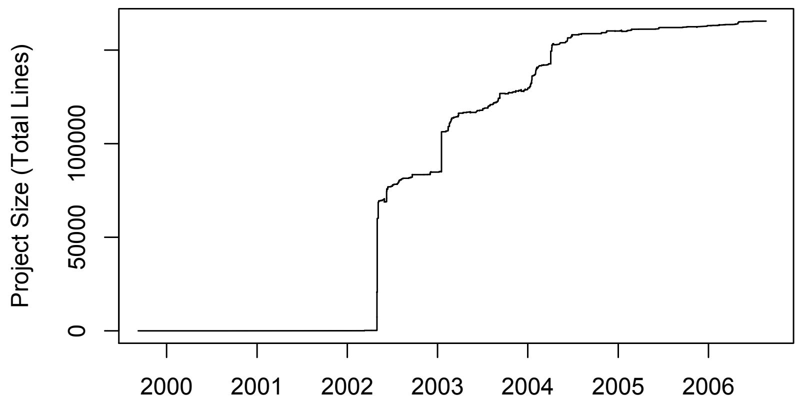


Figure 4.1: Growth of the Java eXPerience FrameWork over time (*all* 160,946 lines of "code").

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

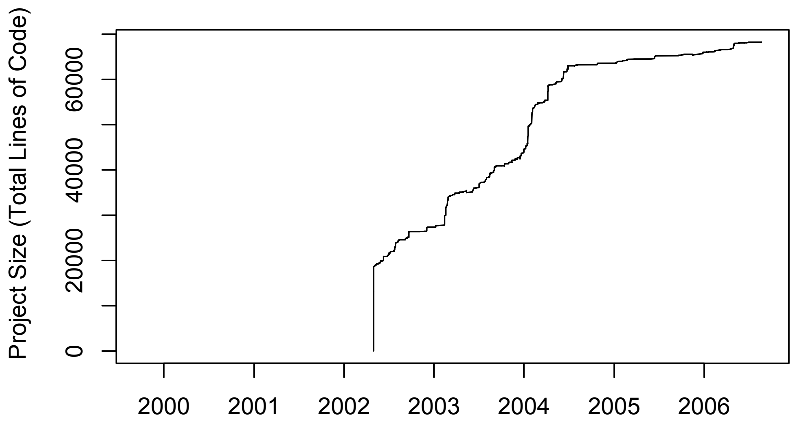


Figure 4.2: Growth of the Java eXPerience FrameWork over time (non-source and auto-generated lines excluded).

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 CVS *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—and in fact, .mdl files were excluded from the original study by filtering file extensions. However, in many projects auto-generated files are legitimate source code and do not exhibit any telltale characteristics. For example, Java user interface code is often generated by tools rather than written by hand.

In the original language fragmentation study, the authors reported auto-generated code as a potential threat to validity and attempted to control for it by running two statistical analyses, one including large commits ($> 5,000$ lines of code) and one *excluding* large commits. The two analyses produced statistically indistinguishable results, and so the authors concluded that the presence of auto-generated files is not an issue in the data set relative to the language entropy calculation.

Nevertheless, auto-generated code, even in smaller quantities—quantities $< 5,000$ lines of code, which are not controlled for in the original study and which we find to be difficult to identify—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 effects likely impact the original calculations and may, for instance, artificially contribute to the observed disparity between single-language-use and multi-language-use groups in the original study.

Internal Development

Another possible cause of cliff walls is internal development—i.e., large quantities of source code developed outside of SourceForge and later committed in bulk. Such activity may result from corporate sponsorship or co-located developers who find it easier to collaborate locally. New source code commits are also often restricted shortly before the release date of a project version, during which time only bug fixes are allowed. Such intermittent stages of restrictive commits may cause periodic cliff walls.

In such cases, SourceForge essentially becomes a distribution tool rather than a collaboration environment. In fact, 12.2% of SourceForge projects (1,221 of 9,997) were only active on a single day, and 50% of projects (5,004 of 9,997) have fewer than 17 active days. One project, "ipfilter" was active for only two and a half hours on August 6, 2006, in which 71,878 lines were checked in; no other changes were made to the "ipfilter" project from that time until the data were extracted for archival four months later.

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.

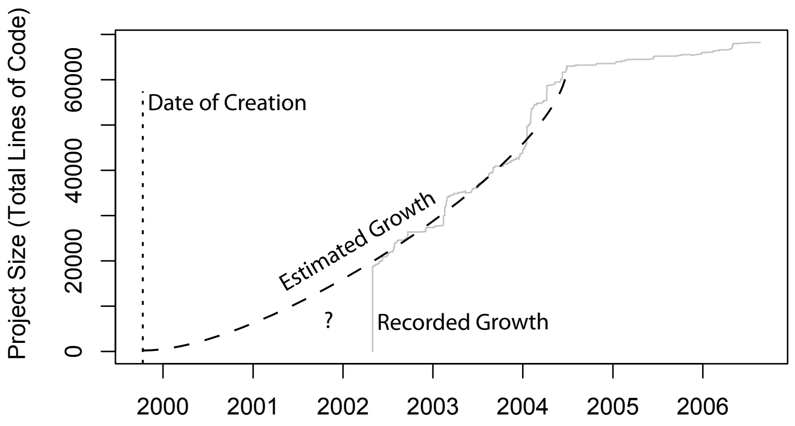


Figure 4.3: Growth of the Java eXPerience FrameWork over time with an overlaid estimate of actual growth (non-source and auto-generated lines excluded).

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.

4.2.3 Generalizing Cliff Walls

Cliff walls as 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, and is only slightly less pronounced in the other three (see Figure 4.5).

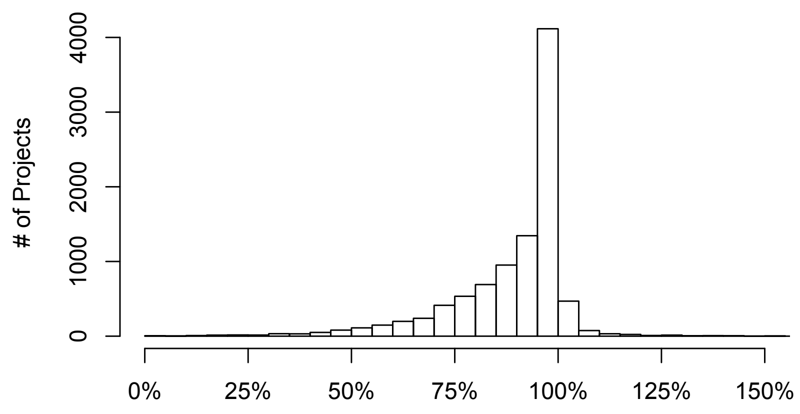
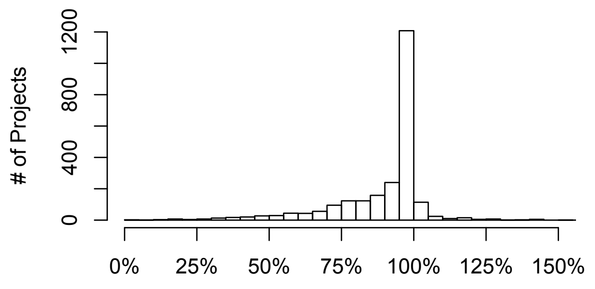
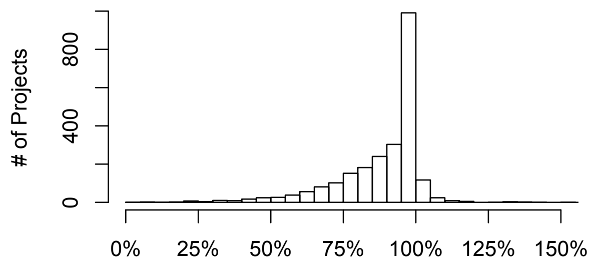


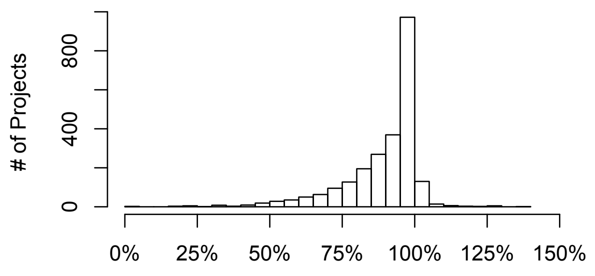
Figure 4.4: Percentage of project size explained by initial commits.



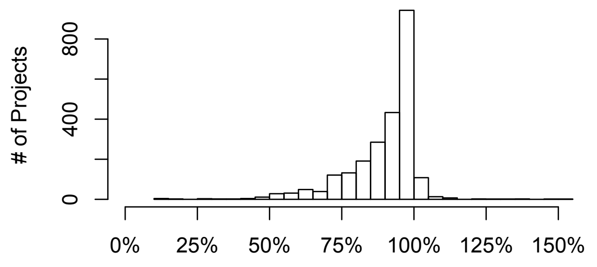
(a) First Quartile (2 to 3,661.25 LOC)



(b) Second Quartile (3,661.25 to 15,027 LOC)



(c) Third Quartile (15,027 to 54,688.25 LOC)



(d) Fourth Quartile (54,688.25 to 27,283,364 LOC)

Figure 4.5: Percentage of project size explained by initial commits. Projects are separated into quartiles by project size (measured in lines of code).

4.2.4 Small Projects

Though some projects are large—such as JXPFW, which was selected as an example in part for its size—the simplest explanation for cliff walls in the average project may simply be small project size. For example, if a 4,000 line daily commit is made to a project with a size of 8,000 lines, that commit represents 50% of the project size. However, the same daily commit to a project with a project size of 500,000 lines would appear negligible. In the sample, 78% of projects (3,197 of 4,094) have a total project size less than 10,000 lines.

One possible explanation for so many small projects in the sample, despite filtering the data for projects declared as "Production/Stable" or "Maintenance" on SourceForge, is the fact that many projects on SourceForge are developed and maintained by a single author. Specifically, 57% of projects (5,688 of 9,997) have only a single author, and projects that can be developed by a single author are generally smaller than those developed by a dozen or more authors. Further, a single author has no need for collaborative tools, and may therefore be less likely to, as they say in agile development, "commit early, commit often." Thus small projects may exhibit more unique author-specific peculiarities, which may not be relevant in a discussion of collaborative product development.

4.3 Author Behavior Pitfalls

In addition to the pitfalls of project data, as discussed in the previous section, we also observe several problems within the developer context that make analysis of language fragmentation using SourceForge data difficult. The original study [57] identifies several limitations, including Marginally Active Developers and Non-Contributing Months, to which we add a third, Author Project Size Bridging. In this section we explore these three limitations in reference to the *cliff walls* phenomenon.

4.3.1 Marginally Active Developers

Krein, et. al. [57] state that marginally active developers—those who contribute small amounts of 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, the data for these authors consist of only a few points and, therefore, are less likely to generate realistic regression lines in the random coefficients model. Therefore, in addition to a separate analysis of the data with only active developers (as suggested in the original study, and which would be interesting), we also suggest using a finer measurement granularity to mitigate the effects of low productivity in the full analysis. If developers are truly developing in multiple languages concurrently, a finer granularity than one month should reduce statistical sampling error (i.e., increase the number of data points) for marginally active developers, while still capturing language fragmentation.

4.3.2 Non-Contributing Months

In addition to marginal activity, many developers do not contribute regularly. Krein, et. al. [57] recognize the potential for problems with this type of data masking and filter the data to remove abnormally large commits. However, given the cliff walls exposed in Section 4.2, the original, naive filters are likely insufficient.

Figure 4.3 shows a time period of two and a half years during which development occurred, but for which data is entirely missing. Were JXPFW's developers selected in the random sample of 500 authors used in the original study, the entire initial development period of two and a half years would have been included in the first analysis as the contribution of a single author in a single month—as if the author *keess* wrote more than 18,000 lines of source code in May of 2002 in HTML, Java, XML, SQL, and CSS. Although the original study ran a second analysis excluding large commits ($> 5,000$ lines of code), it is easy to imagine situations in which large commits fall just below the threshold of exclusion.

Regardless of the size of cliff walls following months of inactivity, they represent a significant unknown variable in the original study, and their actual impact on the results is still unknown. Consequently, researchers must take care to categorize anomalous commits in studies that examine developer commit patterns. Better automated filters are also needed in order to accurately analyze developer activity in large data sets such as SourceForge.

4.3.3 Author Project Size Bridging

Analysis of author contributions reveals that authors do not often bridge between project sizes (i.e., authors tend to contribute exclusively to projects of a similar size). To illustrate, we first discretize projects into groups. Figure 4.6 shows a discretization by quartile on contributions ordered by project size. 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.

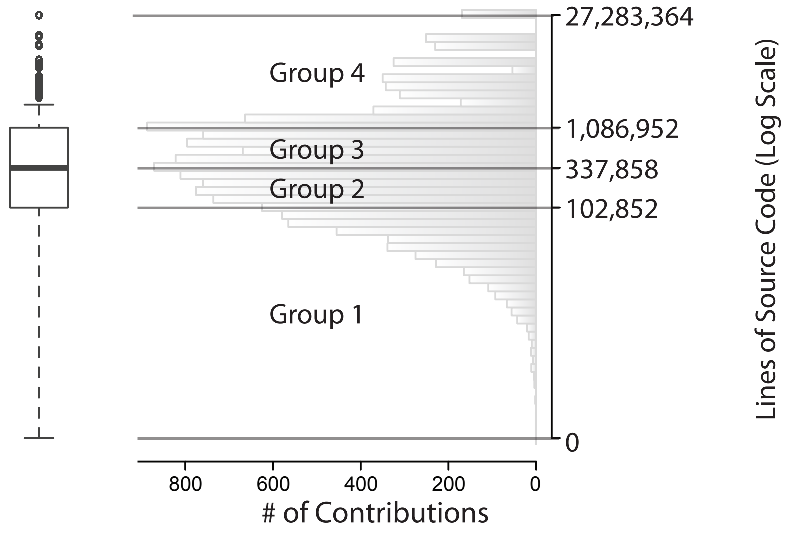


Figure 4.6: Project size groups. Because *lines of source code* is on the log scale (for readability), Groups 3 and 4 cover a much greater range than Groups 1 and 2. The box plot also demonstrates that most of the data in Group 4 are outliers.

As shown in Table 4.1, if projects are discretized into groups using these quartiles, only 6.82% of authors (1,499 of 22,095) contribute to projects in multiple groups, of which 3.66% bridge only between two contiguous groups. Thus 93.18% of authors contribute exclusively within a single contribution group. 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 by contribution group.

When contribution groups are assigned for only those authors who contribute to multiple projects (see Table 4.2), the contrast is not as extreme. However, only 13.14% of authors (2,891 of 21,990) contribute to more than one project. Thus solving the problem

Group	Group Authors	Group Percentage	Combined Authors	Combined 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.

Group	Group Authors	Group Percentage	Combined Authors	Combined Percentage
1	1,197	41.4%	1,392	48.15%
2	75	2.59%		
3	112	3.87%		
4	8	0.28%		
1, 2	633	21.90%	804	27.81%
2, 3	133	3.91%		
3, 4	58	2.01%		
1, 3	340	11.76%	577	19.27%
2, 4	151	5.22%		
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).

by removing single project authors, which sacrifices more than 85% of the data, is not a reasonable solution.

4.4 Other Limitations in the Original Study

In the original study [57], the authors use *lines_added* as the metric for productivity. They argue that *lines_added* captures developer productivity in two ways: first, all new lines committed to a project are recorded by the metric, and second, a line modified is recorded as a *line_added* and a *line_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 initial size represents development effort that was performed outside of the purview of CVS and is recorded exclusively in a separate variable, *initial_size*. Initial sizes represent the vast majority of the size of most projects.

As a result of the omission, the original analysis was calculated on a small, potentially 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 could inflate language entropy, requiring small changes to numerous files. For example, a web developer who adds a field of information to an e-commerce site may make single line changes in SQL, Java, CSS, and HTML files. In initial development, on the other hand, a single developer may work on Java files 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 of the original study.

4.5 Insights and Conclusions

While we have tried to articulate a series of caveats with respect to the use of SourceForge data for evolutionary analysis, we do not intend to send an overly negative message. With proper awareness and appropriate adjustments, SourceForge can still be a good source of data from which to draw useful conclusions about open source development. However, these conclusions must be tempered based on the problems identified in this paper.

4.5.1 Mitigation of Project Problems

Cliff walls occur in a high percentage of projects, prompting several questions that must be answered in order to create effective automated filters:

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 on 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?

4.5.2 Mitigation of Author Problems

Author problems may be slightly easier to overcome than project problems. Unlike project data, author data comes from a single source (although there is some question whether or not source code commits always represent a single developer). Temporal analysis of a developer's activities reveals unusual spikes in development, such as those at the beginning of Figure 4.7, which the original study filtered (individual commits $>$ 5.000 lines of code). After the initial spike it appears that 'keess' has a somewhat normal commit pattern that could be used for analysis. However, further work is required to ensure that this assertion is correct.

4.5.3 Impact on the Original Study

The original study likely suffers from inflated language entropy numbers due to cliff walls and the exclusion of the *initial_size* values. As discussed in Section 4.4, these omissions

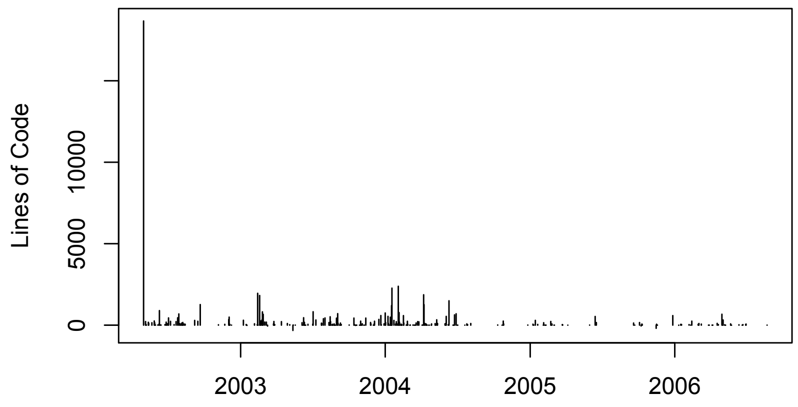


Figure 4.7: Development behavior of author 'keess.'

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 replicated with corrections for these two threats to validity.

4.5.4 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 in a different context—that of project development rather than individual developer activity—it becomes apparent that the original study missed 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. Differentiated replication enables researchers to notice new context variables affecting a study, to which they may otherwise be blind; thus replication is a valuable tool for broadening understanding of a domain.

Chapter 5

Trends That Affect Temporal Analysis Using SourceForge Data[1]

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 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. [69, 25]

[1]This chapter is published in the proceedings of the *5th International Workshop on Public Data about Software Development*, 2010 [63]. Although the content remains unchanged, the text of this chapter has been updated since publication for inclusion in this thesis. Also note that the discussion of *cliff walls* in Section 5.2.2 is largely repeated from the last chapter and may be skimmed.

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 [15]. 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., [12, 26, 41, 56, 68, 88, 94]). 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 [78], recent studies suggest that the two may not be as different as originally thought [13, 32]. Further, as one of the largest OSS hubs, SourceForge hosts thousands of projects—providing extensive data on thousands of mature projects [28]. 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.[2]

Our concerns regarding the limitations of SourceForge data originated from efforts to replicate the results of a previous study [56, 57, 62]. 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 first appear, and that measuring project evolution in SourceForge is fraught with complications. Addressing these limitations is critical 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: 1) non-source files, 2) cliff walls, and 3) 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 exam-

[2]The number of subscribers to the SRDA (SourceForge Research Data Archive) currently exceeds 100 [3]. The actual number of researchers engaging SourceForge data is likely several times that.

ine 9,997 Production/Stable or Maintenance phase projects stored in CVS on SourceForge (archived in October of 2006 [26]).

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. Therefore, studies of source code development should filter by file extension, limiting themselves to source code files that make sense in the context of the particular study.

In order to accurately analyze author and team contributions to projects, we filter out non-source files. Most file extensions occur infrequently in SourceForge data. Of the 21,125 unique file extensions identified, we classify 195 as source code extensions. From this point on, our discussion of project data is limited to recognizable source code based on file extension.

5.2.2 Cliff Walls

Many projects in the SourceForge 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 in order to identify cliff walls programmatically.

The average size of the largest cliff wall within each project is 41.8% of the total size of the project. The median is 30.8%, meaning that half of the projects in the SourceForge data set have a cliff wall that is nearly a third of the project size. Figure 5.2 shows the distribution of SourceForge projects by largest cliff wall as a percentage of total project size (as of the date of data collection).[3] In other words, the histogram represents the number

[3] We removed one outlier from the data set when creating this image. 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

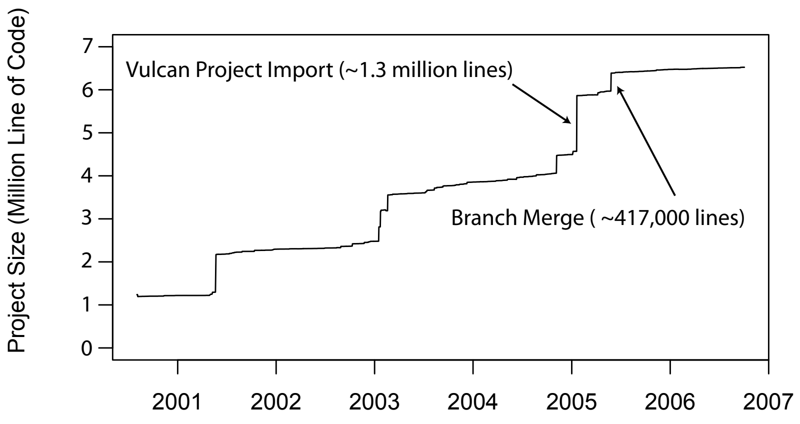


Figure 5.1: Growth of Firebird project over time (all source code).

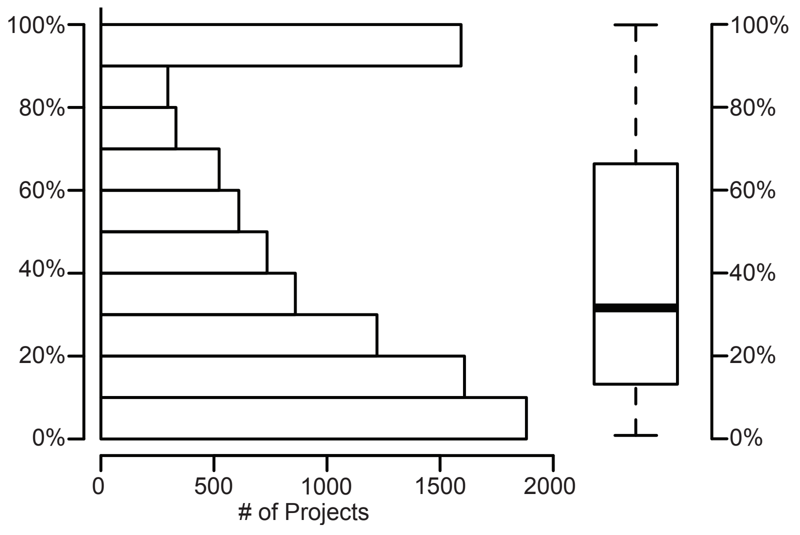


Figure 5.2: Distribution of projects by largest cliff wall, as a percentage of the total project size (one outlier has been removed)—e.g., for the median project (see the associated box plot), the largest cliff wall accounts for just over 30% of the total lines of code.

of projects discretized by their largest cliff wall. For example, in the $0 - 10\%$ bin there are 1,882 projects, meaning that the largest cliff wall in each of those 1,882 projects accounts for between 0 and 10% of the total project size.

Cliff walls appear in all phases of project growth. In Figure 5.1 we see monolithic commits throughout the life cycle of the project. However, in the Java eXPerience FrameWork project (JXPFW) shown in Figure 5.3, the pattern appears only at the beginning of

project was only 4,530 lines of code as of the date the data were gathered. As a result, the project has a somewhat misleading cliff wall percentage of about 313%. All other projects in the data set lie between 0% and 100%.

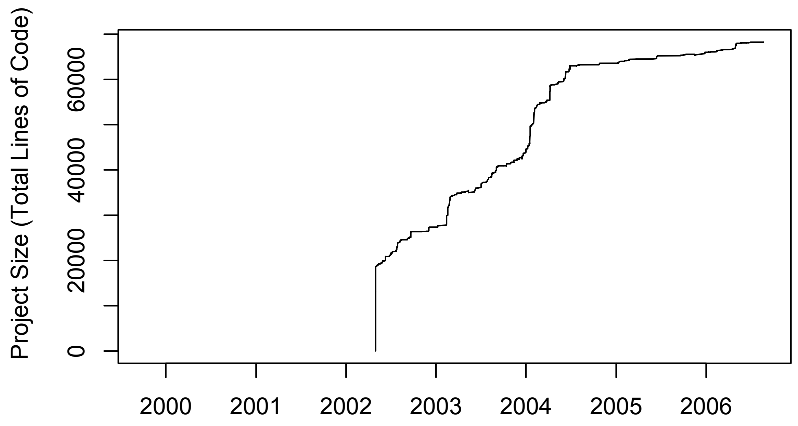


Figure 5.3: Growth of the Java eXPerience FrameWork source code over time.

the project life cycle. JXPFW appears to grow normally after the initial source commit (the cliff wall in 2002, two and a half years after the project was created).

If not accounted for, cliff walls can potentially cause severe bias in analyses of project evolution. For example, if a large commit comprises several months of software development activity, productivity metrics will be erroneously high for the time period just prior to the commit, and developers will wrongfully appear to be inactive for the previous time periods.

Cliff walls appear in project revision histories 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 the SourceForge data set grow almost exclusively via new file commits (as opposed to the modification of files already checked into CVS). The size (in lines of code) associated with new file commits is recorded by CVS separately from lines of code committed to (or deleted from) a preexisting file.

Initial Commit Percentage (ICP) is the percentage of the total size of a project that is made up of initial (i.e., new file) 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 general power law distribution of project sizes on SourceForge [42] (see Figure 5.5) and the assumption that a big commit to a smaller project has a more pronounced effect

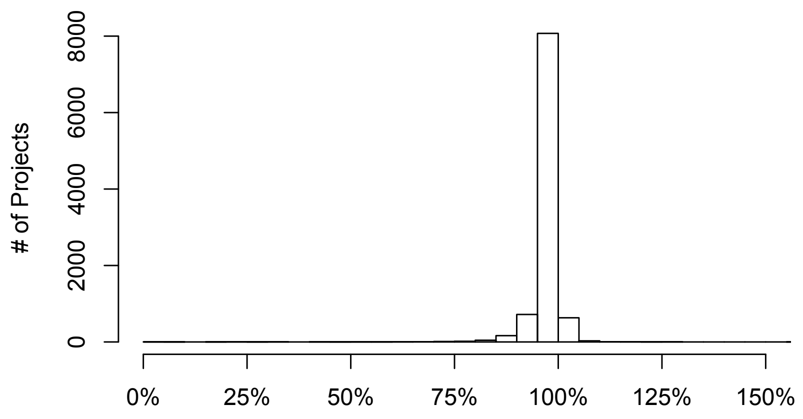


Figure 5.4: Distribution of projects by initial commit percentage.

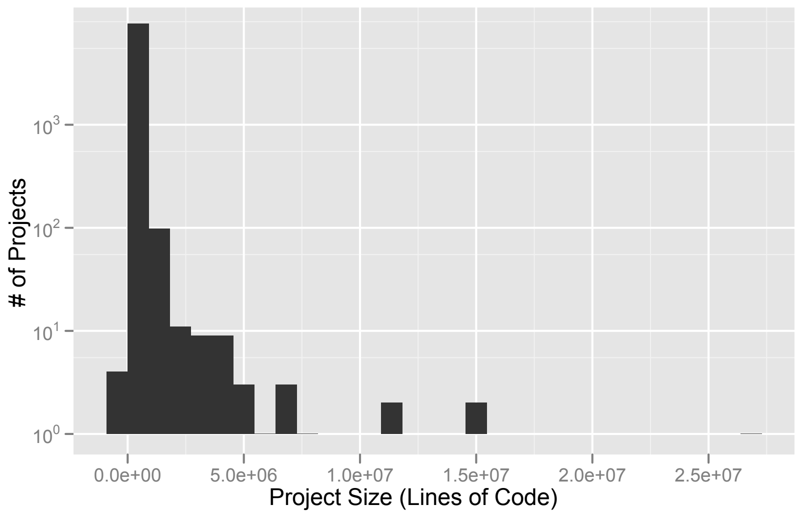
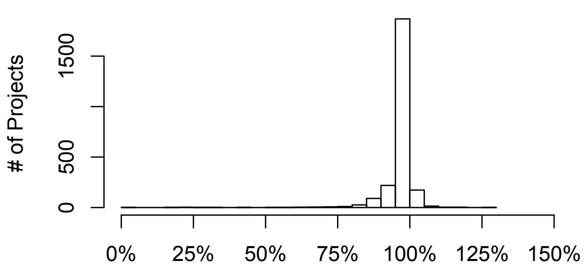


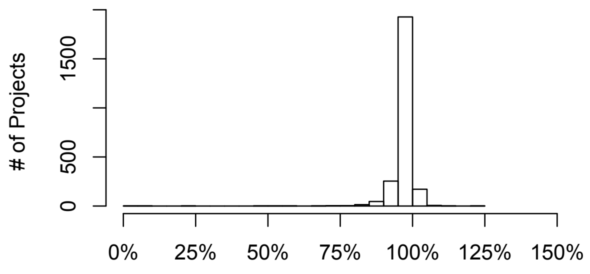
Figure 5.5: Project sizes (log scale on y-axis).

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

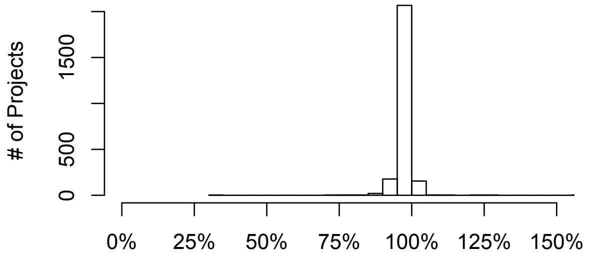
High ICP does not, by itself, threaten appropriately calibrated analyses. However, many of the causes of high ICP may introduce threats to validity, as discussed in Section 5.3.



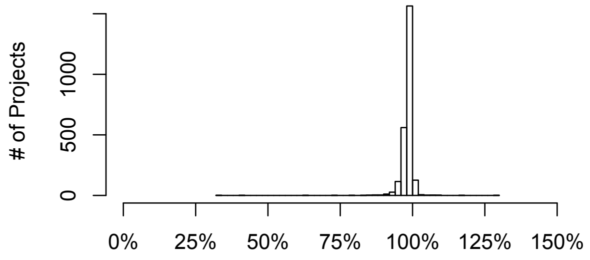
(a) First Quartile (0 to 12,307 LOC)



(b) Second Quartile (12,307 to 58,517 LOC)



(c) Third Quartile (58,517 to 271,848 LOC)



(d) Fourth Quartile (271,848 to 117,147,667 LOC)

Figure 5.6: Distribution of projects by initial commit percentage, separated into project-size quartiles (quartile ranges selected such that 25% of projects fall into each quartile).

5.3 Problem Sources

Although we find many possible causes for the anomalies mentioned in Section 5.2, we identify four specific factors that we consider primary contributors: 1) off-line development, 2) auto-generated files, 3) project imports, and 4) branch merges. Our discussion of these four factors should not be construed as dismissive of other factors. Instead, these four causes represent the largest contributors to the aforementioned anomalies in projects on SourceForge *as a whole*. Within the context of specific individual projects, other factors may turn out to be more significant than these four.

5.3.1 Off-line (Internal) Development

Many projects in the SourceForge data set are first checked into CVS as nearly completed, monolithic chunks. For these projects, authors tend to commit infrequently after the initial check-in—again, often in large chunks, rather than frequent, incremental changes. In these cases, the CVS source code records capture release history, rather than development history.

Thus these projects use SourceForge as a delivery mechanism and not as a collaborative development environment. We postulate that three key factors explain this phenomenon:

The first factor is one of developer locality. It may be easier or more preferable for co-located developers to collaborate via local tools, such as a locally-hosted source code repository, or via tools that are unavailable on SourceForge, such as GIT. Thus these developer teams setup separate "Repositories of Use" for daily work and use SourceForge as only a "Repository of Record" [43].

Second, projects with large corporate sponsors may be developed primarily 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. First tier authors (i.e., gatekeepers) are responsible for reviewing source code before it can be committed to the repository. In fortuitous cases the second tier author creates a branch (see Section 5.3.4) within the SourceForge CVS repository which the gatekeeper inspects before merging 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 the actual development process occurs outside the purview of the repository, in which case first tier authors appear to have developed all of the code, regardless of who actually made the changes.

Each of these three factors produces commits that are bursty and lossy. Instead of recording events throughout the work period, and thereby retaining finer grained development information, authors commit to SourceForge only at the end of a protracted development effort. Consequently, cliff walls are evident in the data and ICP scores are high.

5.3.2 Auto-Generated Files

The presence of auto-generated code is always a concern when analyzing software development data relative to project evolution and individual/group code production. While we believe that much of the code in SourceForge repositories is written manually, developers do use software tools to automatically generate source code (e.g., GUI design tools, lexical analyzers, and program translators [67]). Code generation tools can (and do) produce large quantities of code quickly, which is attributed to whomever commits the auto-generated code. The result is that metrics such as project size, productivity, cost, effort, and defect density are often inaccurate [67]. We believe that commits containing auto-generated code contribute to the presence of cliff walls in the SourceForge data set.

Uchida et al. [92] 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. 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, using code clones to identify auto-generated source code is unlikely to become a panacea.

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.

5.3.3 Project Imports

Figure 5.7 displays 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 the *Firebird* web browser. Imports represent development that occurred not only outside of the repository (as with internal development), but also outside of the project—usually one project being merged into another. Depending on their size, they can result in cliff walls

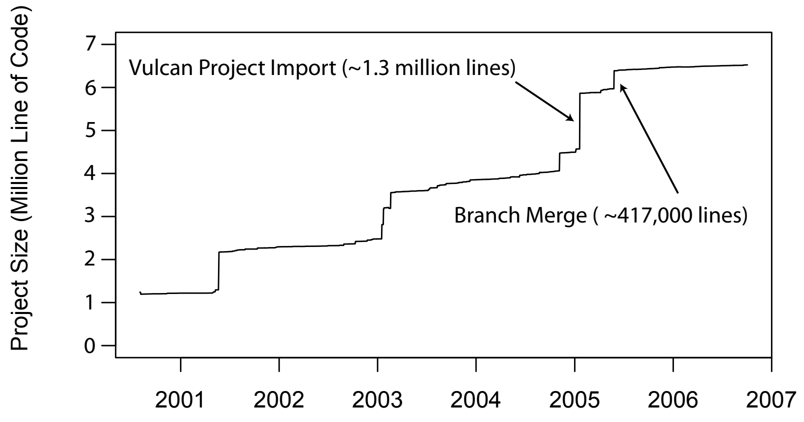


Figure 5.7: Growth of Firebird project over time (all source code; repeat of Figure 5.1).

and high ICP. All code committed through an import is considered an initial revision, rather than a revisionary change.

5.3.4 Branch Merges

The CVS version control system supports branching, a feature that enables concurrent development of parallel versions of a project. However, Zimmermann, Weißgerber, Diehl, and Zeller [95] note that branch merges in CVS cause undesirable side-effects for two main reasons: 1) they group unrelated changes into a single transaction, and 2) they duplicate changes made in the branches. In light of these side effects, many cliff walls can be explained by branch merges:

First, a merge combines into one transaction all transactions on a branch. If a significant amount of development has taken place prior to the merge, the merge will likely appear as a large cliff wall. For example, in Figure 5.7 the cliff wall labeled "Branch Merge" is a merge, not new code. Second, the duplication side effect surfaces when attempting to estimate project size using 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. Finally, branch merges can also falsely inflate measures of author contributions. All of the changes carried over by a branch merge are attributed to the developer who performs the merge, regardless of whether or not that author actually produced those changes. If researchers do not account for branch merges, analysis results may be unreliable.

5.4 Solutions

In order to derive useful, accurate results in temporal analyses of projects hosted on SourceForge we must develop new methods to mitigate the problems identified in this paper. Fortunately, for most of the issues, complete or partial solutions are available that are computable in polynomial time. Nevertheless, for some of the issues, a scalable solution is not readily apparent.

5.4.1 Identifying Branch Merges

In Section 5.3.4 we discuss some of the difficulties that branch merges create for those studying SourceForge data. However, certain approaches may allow researchers to overcome issues caused by merges.

Zimmermann et al. [95] suggest a very simple approach to identifying branch merges: researchers manually examine each transaction for a log message containing the word "merge" and then determine if the transaction is indeed the merge of a branch. Though manual approaches are generally more accurate than automated methods, in this case a manual approach includes significant drawbacks. First, the percentage of actual branch merges that include the word "merge" in the log message is unknown for the SourceForge data set. Consequently, 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, such as the SourceForge data set. Fischer, Pinzger, and Gall [37] suggest a different approach for identifying branch merges in CVS. Their method utilizes revision numbers, dates, and diffs between different revisions of a source file. Their approach, however, is computationally intensive and, as with the Zimmerman method, may not scale to studies utilizing large data sets.

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. Although it would likely suffer in accuracy, the method is feasible for large data sets, and with the addition of a random manual sampling of projects to estimate the percentage of branch merges in the data set, researchers may find it reasonably accurate (false positives and false negatives may balance out). Otherwise future work is required to establish better methods for identifying branch merges within large data sets.

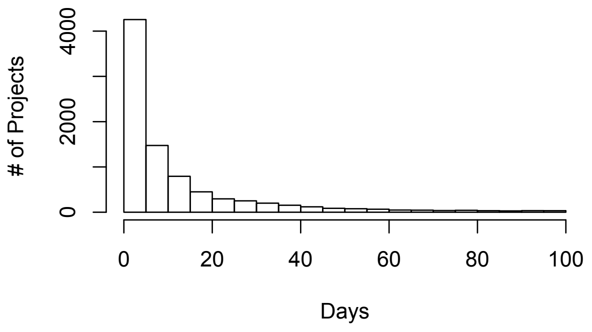
5.4.2 Author Behavior

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."[4] *Frequency of commits* is a metric that captures this behavior. Figure 5.8 displays the distribution of SourceForge projects by commit frequency. We also include in that figure the distribution for projects with more than 40 commits to show that the graphic is not overly biased by small projects, which are completed quickly. According to the figure, there appear to be plenty of projects that satisfy a high-frequency-of-commits requirement. In Figure 5.9 we see that by limiting the data set to projects with more than 40 commits we also filter out most of the short-lived projects.

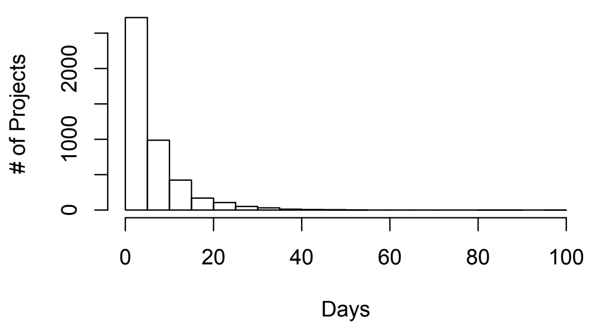
5.4.3 Project Size

Small projects have a much higher occurrence of large cliff walls than large projects. Figure 5.10(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 5.10 have a spike at 100%. However, for Figures 5.10(b), 5.10(c), and

[4]Concept taken from agile software development.

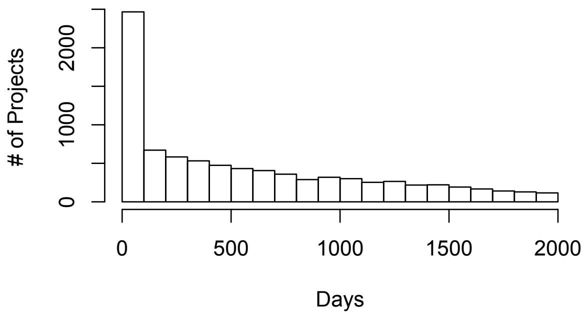


(a) All Projects

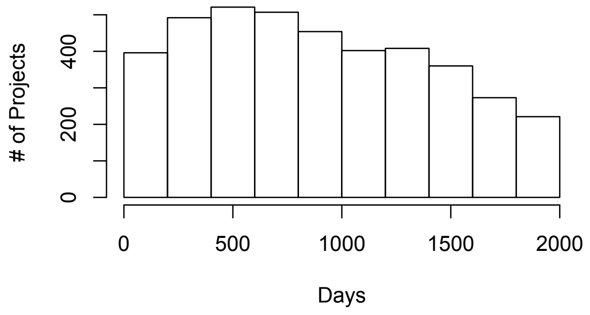


(b) > 40 commits

Figure 5.8: Distribution of projects by frequency of author commits.



(a) All Projects



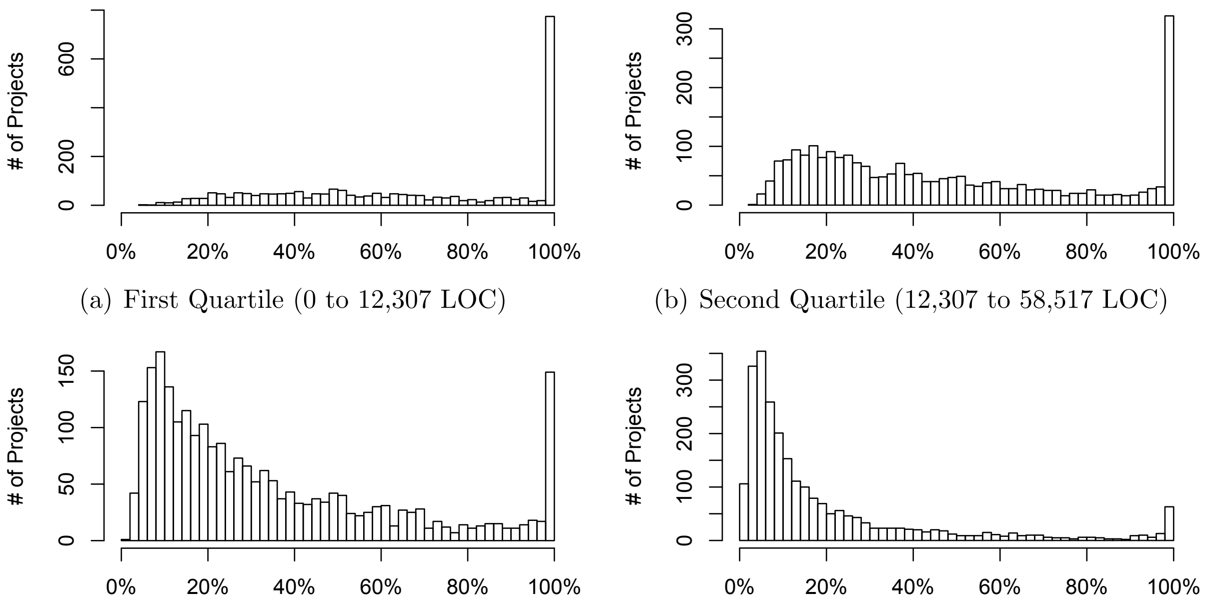
(b) > 40 commits

Figure 5.9: Distribution of projects by project *lifespan* (the time between the first and last commits in a project).

5.10(d), the area under the curve at 0% successively increases. This trend suggests that in the second, third, and fourth quartiles there are many projects that have small, incremental commits and, consequently, that may be appropriate for temporal analysis.

5.5 Insights

Artifact-based evolutionary research utilizing SourceForge data can yield reasonably unbiased results, corroborated by thousands of projects. However, we must choose projects cautiously to avoid measurement error. In order to develop automated techniques for selecting usable projects based on the context of a given study, further work is necessary to classify projects and to create a general taxonomy of the SourceForge ecosystem. Additionally, analysis of the interaction between known project variables may help expose projects



(a) First Quartile (0 to 12,307 LOC)

(b) Second Quartile (12,307 to 58,517 LOC)

(c) Third Quartile (58,517 to 271,848 LOC)

(d) Fourth Quartile (271,848 to 117,147,667 LOC)

Figure 5.10: Distribution of projects by largest cliff wall as a percentage of project size.

that capture a fine-grained development effort. Figures 5.8 and 5.9 suggest that a significant subset of medium-to-large projects on SourceForge can be used for evolutionary analysis. We hope that as project selection methods are further refined, we can develop an automated procedure for choosing projects that have appropriate detail in their revision history.

Chapter 6

A Case for Replication: Synthesizing Research Methodologies in Software Engineering[1]

Software Engineering (SE) problems are—from both practical and theoretical standpoints—immensely complex, involving interactions between technical, behavioral, and social forces. In an effort to dissect this complexity, SE researchers have incorporated a variety of research methods. Recently, the field has entered a paradigm shift—a broad awakening to the social aspects of software development. As a result, and in concert with an ongoing struggle to establish SE research as an empirical discipline, SE researchers are increasingly appropriating methodologies from other fields. In the wake of this self-discovery, the field is entering a period of methodological flux, during which it must establish for itself effective research practices. We present a unifying framework for organizing research methods in SE. In the process of elucidating this framework, we dissect the current literature on replication methods and place replication appropriately within the framework. We also further clarify, from a high level and with respect to SE, the mechanisms through which science builds usable knowledge.

6.1 Introduction

Even though a scientific explanation may appear to be a model of rational order, we should not infer from that order that the genesis of the explanation was itself

[1]This chapter is published in the proceedings of the *1st International Workshop on Replication in Empirical Software Engineering Research*, 2010 [55].

orderly. Science is only orderly after the fact; in process, and especially at the advancing edge of some field, it is chaotic and fiercely controversial. [79, p. 108]

Nancy Leveson [59] introduced her keynote address at the 1992 International Conference on Software Engineering with this quote on the chaotic and controversial processes of science. The observation that science is disorderly and that scientific explanations are not born in coherence connects deeply with researchers and practitioners of Software Engineering (SE), who for decades have struggled to study the immensely complex phenomena by which software practitioners create some of the most complex systems ever conceived by humankind.

In the wake of the tragic radiation therapy overdoses by the Therac-25 (in the late 1980s) [60] Leveson's 1992 observations foreshadowed the catastrophic failure of the Ariane 5 rocket in 1996 [31]. In both cases, software development processes were shown to have been the cause. These events, among others, helped to bring about a broad awakening in the SE research community—a general recognition of the immature scientific foundations on which practitioners were building software. Referring to those foundations, Leveson further noted that "we may be straining at the limits of what we can do effectively without better inventions based on known scientific and engineering principles" [59, p. 7].

Although several researchers, particularly those at the University of Maryland [8, 86], had at that point been working for several decades to establish sound research methodologies within SE, as a field, SE had not yet broadly understood, nor incorporated their work. Consequently, SE practitioners were, prior to 1990, operating on theoretical principles grounded primarily in anecdotal evidence.

In response to growing concerns regarding the state of research practices in SE, advocates for empirical science began educating the SE community, making the case for analyzing and improving the field's core research methods [7, 9, 50, 51, 59, 74, 90]. Numerous forums were organized to advance the state of empiricism in SE and to incentivize researchers to adopt more rigorous practices [86]. These efforts have significantly improved the empirical foundations of our field.

6.1.1 A Unified Framework for Research Methodologies

Although the SE discipline has progressed significantly over the past fifteen years, it has not yet *fully* synthesized, understood, established, and embodied research principles by which to produce usable knowledge [9, 48, 86]. Many researchers have contributed to the topic of knowledge building [9, 24, 47, 48, 83, 86], but the numerous frameworks, processes, observations, and insights that have been put forth have not been adequately synthesized and unified under a common understanding [86].

Objective 1. *Establish a unified framework for SE research methodologies.*

6.1.2 A Context-Specific Interpretation of Replication

In an effort to separate "what is actually true" from "what is only believed to be true," SE has, for more than two decades, been pushing to establish its empirical foundations [9, p. 456]. The idea behind the move to empiricism is to separate truth from belief, and in so doing to build knowledge. However, we seem to have applied much of our empiricism to an endless set of new problems, such that after several decades, "the balance between evaluation of results and development of new models is still skewed in favor of unverified proposals" [9, p. 456] [47, 90].

To help address this problem, Brooks, Roper, Wood, Daly, and Miller [16, 17, 24] refined for SE a set of principles from other disciplines for conducting scientific replications. They presented the first formal embodiment of the concept of replication to the SE community in the mid-1990s. Since that time the topic has been analyzed by several researchers and various groups have made efforts to incorporate replication practices into SE [48, 49, 61, 82, 83]. Nevertheless, the low frequency of replications (originally noted by Brooks et al. [16]) seems to have only marginally improved [38, 48].

We believe, as Juristo and Vegas have indicated, that "we might be dealing with the issue of SE experiment replication from too naïve a perspective" [48, p. 356]. The traditional purpose of replication (i.e., to validate results in case of experimental flaws or fabrication) needs to be analyzed and adapted to the SE context [48].

Objective 2. *Refine the concept of replication as it applies to SE and place it appropriately within the unified framework of research methodologies.*

6.1.3 An Understanding of the Knowledge Building Process

When effectively practiced, replication has the power to build knowledge [17]. Juristo and Vegas note that "[a]fter several replications have increased the credibility of the results, the small fragment of knowledge that the experiment was trying to ascertain is more mature" [48, p. 356]. However, there is little discussion in the literature on the actual mechanisms and processes by which replication *matures* knowledge, and the obvious function of replication as a guardian against experimental mistakes and fabrication does not really satisfy that question. Further, attempts at "exact" replication, in order to validate results, have thus far proven to be difficult and (in isolation) relatively ineffective [48].

Incorporating the recent work of Juristo and Vegas [48], we attempt to answer the question of how replication can be used effectively within SE to build knowledge. We posit that replication—properly interpreted—is the key to building knowledge and, consequently, the primary mechanism by which theoretical paradigms [58] are created, evolved, broken, and replaced.

Objective 3. *Clarify the mechanisms through which science builds usable knowledge and identify the role that replication plays in the knowledge building process.*

In pursuing these three objectives, we must be clear that none of the component ideas in this paper are new. Our contribution occurs in the synthesis of methods and insights from numerous other works, which we attempt to juxtapose within a governing framework.

Therefore, we provide synthesis, structure, and refinement to concepts that have previously been presented, largely independent of one another.

In the next four sections we lay the groundwork necessary for addressing the outlined objectives. Section 6.2 discusses the complex nature of SE and a theoretical perspective with which to manage that complexity. Section 6.3 identifies three fundamental levels on which we need to simplify research results and discusses the problem of premature generalization. Section 6.4 outlines the primary SE research methods and argues the necessity of a multi-method approach to SE. And finally, Section 6.5 explores the concept of fundamental patterns, the process of discovering those patterns, and the role of replication in that process. Following those four sections, we present the Cycle of Maturing Knowledge, a unified framework for research methodologies in SE.

6.2 Software Engineering is Complex

Modern software systems are immensely complex, but SE has formulated metrics and techniques by which to measure, handle, and dissect that complexity, thus enabling practitioners to produce large-scale systems that were previously infeasible. However, despite advances in software systems design, software production processes have not yielded as completely to dissection and understanding. Researchers have studied these processes for nearly half a century, but still struggle to synthesize observations into general theories that hold consistently across environments. Consequently, during the question-answer session following Steve McConnell's keynote address at the 31st International Conference on Software Engineering [66], when asked for his opinion regarding the implications of specific development techniques, McConnell responded with the oft-repeated SE phrase, "It depends." McConnell then elaborated on the implications of various development contexts and the limitations of SE theory to predict context-specific interactions.

The complexity of software development environments, which arguably exceeds that of the largest software systems (especially when considering the human and social elements), is a significant contributor to the context-specific limitations of SE theory. SE environments—and thus SE experiments—involve deep interactions between technical, behavioral, and social forces. On this point, Juristo and Vegas [48] note that the context of a single experiment is subject to literally hundreds of variables. That claim is easily supported, for example, by the seemingly simple concept of programmer productivity, which has been linked by numerous studies to upwards of 250 contributing factors [46, 72].

6.2.1 Common Theoretical Perspectives for Managing Complexity

In an effort to understand the complexity of software environments, researchers have adopted a number of theoretical perspectives. In most cases, these perspectives have not been explicitly articulated. In fact, for most researchers, the choice of perspective is likely made unconsciously. Nevertheless, maintaining an accurate theoretical perspective is vital to the experimental dissection, synthesis, and interpretation processes. An inaccurate perspective can lead to research bias and general misalignment of evidence.

Two of the most common theoretical perspectives adopted by researchers in fields that straddle both technical and social domains are *determinism* and *social constructivism.*

Determinism

The deterministic approach to managing complexity depends on the belief that the phenomena of interest exist independent of human factors; if the phenomena do intersect with people, then they are only subject to the most basic mechanical forces inherent in the human condition, which are, for the most part, beyond the control of conscious thought. Under this assumption, determinists focus on modeling finite relationships between measurable variables, the underlying goal of which is to understand the cause-effect interactions at play in a system. Understanding cause-effect interactions enables system administrators to optimally configure the system [64, 73].

Not surprisingly, computer scientists consistently espouse a deterministic perspective in their research. However, since computer science is predominated by highly technical problems which are generally sterile of human contact, the deterministic perspective is fairly effective. Modern software development, on the other hand, is a team activity, deeply embedded in social structure and human consciousness. Despite that fact, SE researchers often turn to a deterministic perspective when confronted with the complexity that arises from the social aspects of software development. Responding to this tendency toward a deterministic view of research, Leveson stated, “[W]e need to avoid equating humans with machines and ignoring the cognitive and human aspects of our field” [59, p. 9].

Social Constructivism

From a socially constructed perspective, phenomena are said to be produced in consequence of human decisions and social mechanisms. Therefore, social constructivists seek to explain phenomena through an interaction of human factors. Creating predictive theories requires understanding the motivations, objectives, interests, limitations, and interactions of the human participants [73, 93].

Since most SE researchers come from highly technical backgrounds, as a field, SE is partially blind to the socially constructed perspective of software development. In making that statement, however, we do not want to denigrate the excellent work that many of our colleagues undertake to study social and cognitive factors. Nevertheless, as a discipline, our knowledge and skills in sociological and psychological research are still immature, and the tools that we *have* borrowed from other disciplines have not yet fully disseminated within our science. Despite these difficulties, we are making progress. The most difficult step to take is, in fact, the first—to see the world in a new way.

6.2.2 An Enacted View of SE Complexity

Both determinism and social constructivism are applicable to SE—each contributes fundamental insight into the mechanisms governing software development environments and project ecosystems. However, the two perspectives are diametrically opposed, and as such, they are each only able to capture particular facets of "the big picture." Since both views oversimplify SE phenomena, we propose adapting and incorporating into SE a theoretical perspective originally formulated by Orlikowski and Iacono [73] for studying "the digital economy," termed *an enacted view*.

An enacted view of SE incorporates both deterministic and socially constructed forces into the same paradigm. This perspective maintains that software production is neither a balancing of technical and logistical trade-offs, nor is it simply a matter of managing people. In the following quote by Orlikowski and Iacono we substitute "software engineering" in place of "the digital economy" and note the profound applicability of the enacted view in examining SE:

> This view suggests that [*software engineering*] *is neither an exogenous nor a completely controllable phenomenon*, but an ongoing social product, shaped and produced by humans and organizations, and having both intended and unintended consequences. It is our individual and institutional actions in developing, constructing, funding, using, regulating, managing, supporting, amplifying, and modifying the phenomenon we refer to as ['software engineering'] that enacts it over time. [73, pp. 357–358, italics added]

Assertion 1. *SE represents the culmination of a rich array of interactions between technical, behavioral, and social forces, and only when considering it from that full perspective will an accurate "big picture" begin to emerge from the complexity.*

6.3 The Need to Simplify: Cognitive, Scientific, and Practical

Innately, we all have a need to simplify the world around us. At its most fundamental level, the need to simplify is motivated by core cognitive processes. These processes seek to categorize the physical and abstract elements of our environments into generalized schemas, which are critical to the storage and retrieval of information in memory. The categorization mechanism has been shown, for instance, to be a driving force behind social stereotyping, in which we mentally classify each other based on prominent and identifiable features, such as race, gender, and social status [5]. Research in organizational behavior has also observed that, as participants in an organization, we generalize strategic issues into a hierarchy of opportunities and threats [34]. Thus as humans, we maintain numerous taxonomies by which we organize the elements of our environments.

In addition to taxonomies, which represent elements and element-relationships, human cognition also relies on operational rules to govern the interactions between the elements in our numerous taxonomies [87]. These rule sets, which enable us to make judgments about the world (e.g., to predict the consequences of adding manpower to a late project), are generalizations of cascades of interactions that occur between elements of the real environment.

As previously discussed, SE researchers sometimes oversimplify software environments—for instance, by espousing a deterministic perspective of software production mechanisms. In a notable case, recent evidence [13] suggests that Eric Raymond's *The Cathedral and the Bazaar* [78], oversimplifies the organizational structures of productive open source software projects. However, generalization is not necessarily a detriment to science. In fact, scientific inquiry relies on the distillation of information in order to be productive. Since knowledge is an abstract representation of elements and the relationships between those elements, building knowledge requires constructing taxonomies and rule sets. Science, therefore, is designed to be a conscious and formalized embodiment of the natural cognitive processes by which we come to "know" the world, and as such, relies on the digestion and generalization of environmental complexity. Indeed, the need for generalization in science is reflected throughout the meta-theoretical principles on which it operates—consider, for example, Occam's razor.

In addition to cognitive and scientific needs for generalization, daily life also depends on simplified taxonomies and rule sets. Day-to-day living generally requires us to make judgments quickly—for instance, do I walk or drive to work today? In the practical application of SE we often have more time to make strategic organizational decisions than when we're late for work, but those decisions still must be made in relatively short time frames. Simplified environmental models are the mechanism by which we make efficient decisions.

Assertion 2. *The abstract models by which we interpret and manage environmental complexity must be simplified on at least three levels: cognitive, scientific, and practical.*

6.3.1 The Problem of Premature Generalization

As Leveson [59] explains, intuition is a necessary component of formulating hypotheses. If intuition is an informal theoretical synthesis of prior observations, then hypotheses are the formalized, testable embodiments of that intuition. However, intuition can be misleading, and thus hypotheses must be tested. As such, Leveson boldly states that we "need to recognize the unproven assumptions and hypotheses underlying our current software engineering techniques and tools and evaluate them in the context of what has actually been demonstrated about these hypotheses instead of what we would like to believe" [59, p. 8].

In response to this and similar arguments, SE researchers have conducted numerous empirical studies. However, Basili, Shull, and Lanubile [9]—and more recently Jørgensen and Sjøberg [47]—caution that the expanse of new proposals extends far beyond the breadth and depth of our theories. With so much data now in massive repositories and with computer networks bringing it to our fingertips, it's no wonder that published research is often reduced to a sophisticated game of trivial pursuit. At a certain point, we risk drowning in a sea of "new proposals" and unsynthesized results. Because our studies have become highly disconnected [47, 86]—and since clearly no single study has the independent power to produce definitive results [47, 48, 82, 86]—by definition, all generalizations we can make will be shallow and immature. Without greater research synthesis, therefore, we will continue to ignore the most interesting phenomena of our field, and our theories will remain shallow and narrow in scope.

Assertion 3. *Disconnected research necessarily leads to premature generalization and narrow, immature theories.*

6.4 No Silver Method

According to Popper [75], theories are reliable only after researchers have made numerous attempts to falsify them. Thus, a theory must ultimately be grounded in observation, or it is of little value. Conversely, empirical—that is, observational, experiential, or experimental—work is useless unless the resulting observations are synthesized into governing theories. Empirical and theoretical methods are therefore complementary. Theoretical work involves logic, deduction, inferences, and thought experiments, whereas empirical work requires observation and, when formalized, measurement (qualitative and/or quantitative). In response to the question, "How will we build our [scientific] foundation?" Leveson states, "It will require both building mathematical models and theories and performing carefully-designed experiments" [59, p. 7]. Thus both theoretical and empirical methods are necessary to capture and dissect the complex problems of SE [7, 9, 47, 50, 51, 86].

Empiricism comprises numerous research methods (e.g., controlled studies, ethnographic studies, and data mining and analysis), which can be grouped into two categories: *observational* and *experimental*. Observational methods are conducted "in the wild"—that is, the phenomena are observed in their natural context. Experimental methods, on the other hand, examine phenomena under controlled conditions, in which the researcher selects subjects to receive "treatments" and then observes the consequences. If properly designed, experimental methods enable the researcher to conclude with a degree of confidence that the variation in the applied treatment caused the observed variation in the resulting experimental groups. The key difference, therefore, between observational and experimental methods is whether the researcher controls the context of the study.

If experimental methods enable researchers to make causality inferences, then why perform observational studies at all? In SE the most common answer to this question is that

SE experiments are expensive or impractical to conduct. Although that response is generally accurate, it risks mischaracterizing observational methods.

The goal of empirical work is to observe both the elements of a system and the relationships between those elements in order to build an understanding of the dynamics of that system. Creating an accurate and complete understanding of a system requires that two conditions be met: 1) the understanding must capture and incorporate all elements and element-relationships in the system, and 2) the understanding must completely dissect and digest those elements into their constituent parts. We have already observed that experimental methods are more useful than observational methods for dissecting complexity. However, experimental methods are limited in their scope, since they cannot explicitly address all important variables, many of which are not readily measurable. Experimental science relies on random assignment of subjects to treatment groups to compensate for missing information. If the samples are large enough and truly random, then cause-effect relationships can be confidently identified—*but only those relationships for which the researcher is specifically testing.* Any elements not explicitly identified and designed into the experiment cannot be objectively dissected from the system. Thus at the extremes, research methods either fail to capture the full complexity of the system, or they fail to objectively dissect that complexity. In reality, *the more complexity a method preserves, the less power it has to dissect that complexity.*

Consider, for example, case studies. The primary criticism levied against case studies is that they tend to be anecdotal. How is one to know that a researcher's intuition is accurate? To this effect, Basili, Shull, and Lanubile state, "Experimentation *in software engineering is necessary.* Common wisdom, intuition, speculation, and proofs of concepts are not reliable sources of credible knowledge" [9, p. 456]. We agree with this criticism, but note that it simply echoes the limitation of observational methods, that they are unable to objectively dissect complexity.

Utilizing experimental methods, however, we still fail to satisfactorily generalize our results. Whenever the "it depends on context" qualification is applied to conclusions, then the methodology has abstracted away complexity in favor of dissection. In order to meet both requirements for accurately understanding a system, breadth as well as depth, we must learn how to effectively blend diverse research methods into synthesized analyses. As Frederick Brooks [18] might say, there is no "silver bullet"—or method.

Assertion 4. *No single research method can fully reveal the complex mechanics of SE.*

6.4.1 The Multi-Method Approach to SE: Embodying an enacted view

At present, the best solution to the conflicting limitations of observational and experimental research methods is, as Daly [24] terms it, the *multi-method approach*.[2]

According to Daly, the "multi-method approach involves using two or more different empirical techniques to investigate the same phenomenon" [24, p. 65]. Daly notes that corroborating results (across multiple differentiated studies) provides confirmatory power and represents a form of validation. However, we propose to extend Daly's concept of the multi-method approach by recognizing that its most fundamental and powerful contribution, when applied consistently, is to knowledge building. In fact, Daly hints at this contribution when discussing an evolutionary embodiment of the approach, in which researchers leverage the strengths of various research methods at specific stages in a research program in order to incrementally build their understanding. Each successive stage is designed around the knowledge gained from the previous stages—thus the preliminary results not only impact the final conclusions, but they configure the overall methodology as the study proceeds.

In sections 6.5 and 6.6 of this paper, we discuss the power of fundamental patterns and employ the multi-method approach as a fundamental pattern under which we synthesize

[2]We recognize that the idea of the multi-method approach is not original to Daly and that it has been practiced in research labs since the foundations of SE. We refer to Daly's work because it is the most self-contained and thorough treatment of the concept of which we are aware.

a unified methodology for SE research. In the process of elucidating the overall research framework, we dissect the current literature on replication methods and place replication within that research framework. As a consequence of these efforts, we also further clarify, from a high level, the mechanisms identified by Basili et al. [9] through which science builds usable knowledge.

6.5 Patterns

Although finding interesting phenomena is relatively easy in SE, examining them deeply [80] and placing them appropriately within the context of a plethora of other interacting elements is as difficult as ever. Sometimes, amid the chaos of interactions, finding useful generalizations seems impossible. The difficulty arises from the complexity of SE environments and ecosystems, which are *enacted* over time by technical, behavioral, and social forces and are therefore continually evolving. Drawing again from Orlikowski and Iacono's work on "the digital economy," we suggest that accurately generalizing SE complexity requires a shift in perspective to fully embrace an enacted view, and to embody that view in our research. Using the words of Orlikowski and Iacono,

> [Software] is a phenomenon that is embedded in a variety of different social and temporal contexts, that relies on an evolving technological infrastructure, and whose uses are multiple and emergent. As a result, research studies will yield not precise predictions—because that is not possible in an unprecedented and enacted world—but underlying *patterns* . . . Similarly, research studies will offer not crisp prescriptions—because these are unhelpful in a dynamic, variable, and emergent world—but general *principles* to guide our ongoing and collective shaping of [software]. [73, p. 375]

Thus, we need to structure research methods in SE to enable us to discover the fundamental patterns underlying and interconnecting the observable phenomena. But what are fundamental patterns? And how do we distinguish universal truths from shallow, context-specific poseurs? Our discussion leads us now to Christopher Alexander's "The Timeless Way of Building" [1] and "A Pattern Language" [2], and from thence to Thomas Kuhn's "The Structure of Scientific Revolutions" [58].

6.5.1 In Search of "Double-Star" Patterns

In Christopher Alexander's original work on architectural patterns [2], he identifies three broad classes of patterns: Patterns associated with two asterisks are those that the authors believe to have captured a true invariant, "that the solution we have stated summarizes a property common to all possible ways of solving the stated problem," and "that it is not possible to solve the stated problem properly, without shaping the environment in one way or another according to the pattern that we have given—and that, in these cases, the pattern describes a deep and inescapable property of a well-formed environment." Patterns with one asterisk are those believed to "have made some progress towards identifying such an invariant." Those with no asterisks "have not succeeded in defining a true invariant," meaning that their pattern represents one particular solution, but does not capture the essence of the problem. [2, p. xiv]

Essential invariants involve both the *structure* of the architectural element in question, as well as *the way in which such an element interacts with human behavior*. As you read the following architectural example from Alexander, consider the implications to SE:

> [I]n New York, a sidewalk is mainly a place for walking, jostling, moving fast. And by comparison, in Jamaica, or India, a sidewalk is a place to sit, to talk, perhaps to play music, even to sleep. It is not correct to interpret this by saying that the two sidewalks are the same. . . . Each sidewalk is a unitary system, which includes *both* the field of geometrical relationships which define its concrete geometry, *and* the field of human actions and events, which are associated with it. So when we see that a sidewalk in Bombay is used by people sleeping, or for

> parking cars . . . and that in New York it is used only for walking—we cannot interpret this correctly as a single sidewalk pattern, with two different uses. The Bombay sidewalk (space + events) is one pattern; the New York sidewalk (space + events) is another pattern. They are two entirely different patterns. [1, p. 73]

Alexander's notion of double-star patterns embodies the idea of context-free solutions, principles and patterns that are not invalidated by an "it depends" clause. If the solution to a problem is, "it depends," then the applicable patterns must, per force be narrowed to the context of the specific problem. As researchers we err when we stretch our narrow findings so broadly that we misperceive a solution in a specific context as having captured a broader principle. Although there is nothing wrong with context-specific solutions to context-specific problems, in order to build knowledge, we must understand the broader, higher-level patterns that operate. Again, an example from Alexander:

> Since every church is different, the so-called element we call "church" is not constant at all. Giving it a name only deepens the puzzle. If every church is different, what is it that remains the same, from church to church that we call "church"? [1, p. 84–85]

Such a discussion brings to mind broad-brush appellations such as "open source" versus "closed source" software development processes, as if we really understand what each expression means or why exactly they matter when it comes to engineers cutting code. "A pattern only works, fully, when it deals with all the forces that are actually present in the situation" [1, p. 285].

Our experience suggests that the quest for fundamental, Alexandrian double-star patterns provides a mechanism for encapsulating chunks of knowledge at various levels of abstraction (from narrow and specific to broad and general), as well as a mechanism whereby meta-level patterns may be discovered and understood that describe the generation and interaction of lower-level patterns [52]. Such tools provide the means for better understanding the scientific models that always underlie context-specific empirical research, but which are often either unidentified or unrecognized.

Assertion 5. *Scientific models are more generalizable when based upon fundamental (Alexandrian) patterns; therefore, it is the job of SE researchers to discover and develop fundamental context-free patterns from which practitioners can compose an endless variety of software organizations and practices, each to meet specific, context-sensitive needs.*

6.5.2 Seeing "The Big Picture"

Finding fundamental patterns requires seeing and understanding, within the same context, a broad range of empirical observations. As any puzzle enthusiast knows, it is only when viewing the pieces in context with one another, correctly fitted, that the overall picture becomes obvious.

In SE we have come to broadly recognize the need for empirical methodologies in order to build knowledge (hypotheses are based on intuition, and so must be tested through observation). However, it seems that our efforts to become empiricists have lead us away from deep observational synthesis and theory building—that is, we have *oversimplified* our methodologies. Indeed, for many of the most important questions in SE, we have failed to establish any relevant formalized theories [47]. Consequently, our studies have become disconnected, thus leading to shallow, narrow conclusions that do not generalize well [47, 86]. It is unfortunate that while most academic papers include an empirical study of some sort, we find almost no papers that are purely (or even primarily) theoretical. Where are the papers synthesizing these empirical studies into a unified, cohesive whole? We agree with Jørgensen and Sjøberg who caution that "there are much too few review papers in software engineering trying to summarize relevant research" and "[w]ithout a much stronger focus on [the] theory-development step, we probably will continue to produce isolated, exploratory studies with limited ability to aggregate knowledge" [47, p. 33].

In addition to fragmented observations and premature generalization, a lack of theory building in science leads to an even more subtle, degenerative problem. According to Kuhn, mature scientific disciplines pass through periods of intense community focus, knit together by deeply shared beliefs about how the world works (i.e., paradigms). These periods of focus are followed by periods of broad self-assessment and theoretical upheaval (i.e., scientific revolutions) [58]. Therefore, science is a two-fold process of evolving theories based on a continual stream of new observations, followed by replacing theories when they inevitably outlive their usefulness. In the words of Greenberg, we must allow for periods of both "getting the design right" (science under a paradigm) and "getting the right design" (scientific revolution) [39, p. 115]. The two are separate and distinct activities, and in the absence of either, scientific inquiry is severely handicapped.

Since theories are the embodiment of a way of thinking, and paradigms form around a common way of thinking, then without theory building, paradigms of broad reach and significant impact cannot occur. As a result, the research community is not cohesive, communication between researchers is difficult, and research findings are disjoint and disconnected. Thus, theory building is necessary for establishing paradigms, and paradigms are vitally important to building knowledge. Without theory building, SE will struggle to maintain deep community focus on a particular set of ideas [9, 58, 86].

To this point, we have identified three issues that prevent researchers from seeing "the big picture"—fragmented observations, premature generalization, and an inability to sufficiently focus community efforts—and have hinted that theory building helps to resolve all three. However, we further observe that theory building is not only necessary to discover fundamental patterns, but it is also *not* sufficient. Theory building is the glue that binds observations together into a cohesive and meaningful context (without which empirical work remains fragmented and context-specific), and that binds a community of researchers together so that theoretical proposals can be deeply probed for limitations. Nevertheless, theories rely on observation for validation and evolution, and so theory building without a rigorous empirical counterpart generally produces beautifully useless "knowledge." Therefore, the presence of both theoretical [47, 84, 85, 86] and empirical[35, 50, 51, 74, 86, 90] methodologies—with an *effective dialogue* between—is necessary in any science in order to discover fundamental patterns.

Assertion 6. *Discovering fundamental patterns requires seeing "the big picture," which is accomplished in science by an alternating process of observation and theoretical synthesis.*

6.5.3 A Terminological Aside

Before we proceed with the discussion at hand, we must clarify some terminology relative to replication methods in SE. In the literature, researchers have employed various terms to describe replication studies, including: *exact* versus *non-exact* [17, 48, 61, 83], *close* [48], *conceptual* [83], and *literal* versus *theoretical* [61].

In selecting our terminology, we consider the two fundamental goals of replication, as outlined by Shull, Carver, Vegas, and Juristo [83]:

1. "Testing that a given result or observation is reproducible" [p. 212].
2. "Understanding the sources of variability that influence a given result" [p. 213].

Generally speaking, goals represent the *intent* of an undertaking, and since the goals of replication are disjoint—meaning that a single replication study cannot address both goals simultaneously—"intent" is a good candidate for a classifier. In the literature, two terms effectively describe the intent of a replication study, *strict* [9] and *differentiated* [48]. A strict replication is one that is meant to replicate a prior study as precisely as possible, whereas a differentiated replication intentionally alters aspects of the prior study in order to test the limits of that study's conclusions.

As a classification scheme, these terms are objective as well as inclusive: objective because the goals they represent are disjoint, and thus no replication study can fit both categories; inclusive because every replication study fits into one of the two buckets. As an added benefit, this classification scheme records the original intent of the study, which is generally far more difficult to deduce from lab packages and published reports than most other attributes of the replication (e.g., its "closeness" to the original study).

In addition to the terms *strict* and *differentiated*, we also adopt the terms *dependent* and *independent* [49, 83], but adjust their definitions slightly. We define a dependent replication to be a study that is specifically designed with reference to one or more previous studies, and is, therefore, *intended* to be a replication study. An independent replication, on the other hand, addresses the same questions and/or hypotheses of a previous study, but is conducted without knowledge of, or deference to, that prior study—either because the researchers are unaware of the prior work, or because they want to avoid bias. When the results confirm prior findings, independent replications can support more robust conclusions than dependent replications. However, contradictory independent replications are generally more difficult to synthesize with prior work, and in most cases, are inadvisable when studying poorly understood phenomena because they make traceability difficult [83, 48].

Thus we arrive at a high-level taxonomy of replication (see Figure 6.1), which embodies the fundamental pattern of intent and is, at all levels, both objective and inclusive.

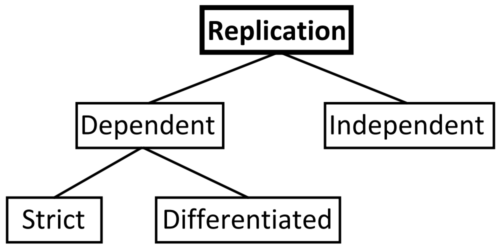


Figure 6.1: High-level replication taxonomy.

6.5.4 Replication: The Key to the Knowledge Building Process

As discussed earlier, the complexity of SE necessitates a multi-method approach to research, which, by definition, requires multiple studies to repeatedly examine the same questions and hypotheses. The idea is (as Basili et al. [9] describe, in terms of parsimony) to tackle threats to validity through many studies, each with different threats, rather than trying to hopelessly control all threats within a single study. If we accept the "parsimony" approach to research, then the alternating process of observation and theoretical synthesis ultimately lies in the domain of replication.

However, replication has traditionally been defined in the literature in terms of *strict* replication, and so researchers have invested significant resources attempting to exactly replicate studies—a practice which has thus far proven to be infeasible in SE. For this reason, Juristo and Vegas suggest that researchers in SE should relax the demand for strict replication, suggesting that "opening the door to non-identical replications could in actual fact encourage researchers to do more replications . . . and, at the same time, [turn] up new knowledge" [48, pp. 365–366]. In fact, since it is unlikely that a replication can be exact, strict replications generally end up being treated like differentiated replications anyway, except that traceability is more difficult because the changes are unplanned.

Although strict replication is important for validating observations—that is, confirming that we accurately understand the conditions under which a set of observations occur—it does not test theory, and therefore, it does not build *usable* (i.e., practical) knowledge. In making this claim, we do not argue that attempting strict replication cannot uncover new information which can in turn be used to build knowledge. We simply observe that strict replication is designed to verify that a study's reported conditions do in fact produce the reported observations. Testing the correlation between conditions and observations to detect experimental flaws or fabrication is not the same as testing the validity of theoretical axioms. Differentiated replication is the mechanism that tests the limits of theoretical axioms by altering the conditions or the context of prior investigations. Strict replication simply tells the researcher whether or not the latest observations should be accepted and allowed to affect the current theory. As an example, consider the work of Lung, Aranda, Easterbrook, and Wilson who, after performing a strict replication, state: "On reflection, the literal replication was much more complicated than we expected, and told us very little about the underlying theory. On the plus side, we identified a number of flaws in [the researcher's] experimental design, which we were able to correct" [61, p. 200]. Therefore, we highlight a somewhat subtle distinction between the type of knowledge created by the practice of validating observations versus that of refining theories. The former practice generates knowledge useful only to the researcher/theoretician, whereas the latter generates knowledge for building theories, and is, consequently, ultimately useful to the practitioner. In case these two assertions appear to be reversed, we further note that in mature scientific domains, practitioners operate on well-established theories, not the results of individual studies.

Assertion 7. *Strict replication is the process by which researchers test a study's methods and procedures to validate that the reported conditions produce the reported observations.*

Assertion 8. *Differentiated replication, founded on the multi-method approach to research, is an embodiment of the alternating process of observation and theoretical synthesis, and is, therefore, the process by which theoretical paradigms are created, evolved, broken, and replaced; consequently, differentiated replication is the primary mechanism by which fundamental patterns can be discovered and is the key to making the knowledge building process productive.*

6.6 The Cycle of Maturing Knowledge

Having laid the necessary groundwork, in this section we present a unified framework for research methodologies in SE, which we refer to as the Cycle of Maturing Knowledge (CMK). We first summarize the philosophical constructs presented thus far, after which we discuss the CMK in detail.

In SE we struggle to produce usable knowledge that generalizes across software environments. We struggle in part because we have not yet developed sufficient methods to manage the interacting technical, behavioral, and social complexities of modern software development. Currently, our empirical studies are disconnected and our theories are generally shallow. In order to overcome the problem of premature generalization, we must embrace more fully the complexity of SE. Guided by an appropriate theoretical perspective (e.g., an enacted view), and through an alternating process of observation and theoretical synthesis, we can learn how to perceive "the big picture" amidst the complexity. With the requisite community focus, we can further discover fundamental patterns, the essence of which must form the structure of our theories. Based on fundamental patterns, our theories will be more generalizable, enabling us to distill usable knowledge.

The CMK (depicted in Figure 6.2) incorporates each of the elements discussed in this paper and represents a SE-specific adaptation of the general knowledge building process discussed in the literature [9]. The process begins with a common experience: an epiphany or insight gained from preliminary observations of some set of phenomena. This experience generally progresses through four phases: 1) we observe an aspect of the world; 2) our observation triggers a new insight; 3) we begin "seeing" the concept embodied by our insight

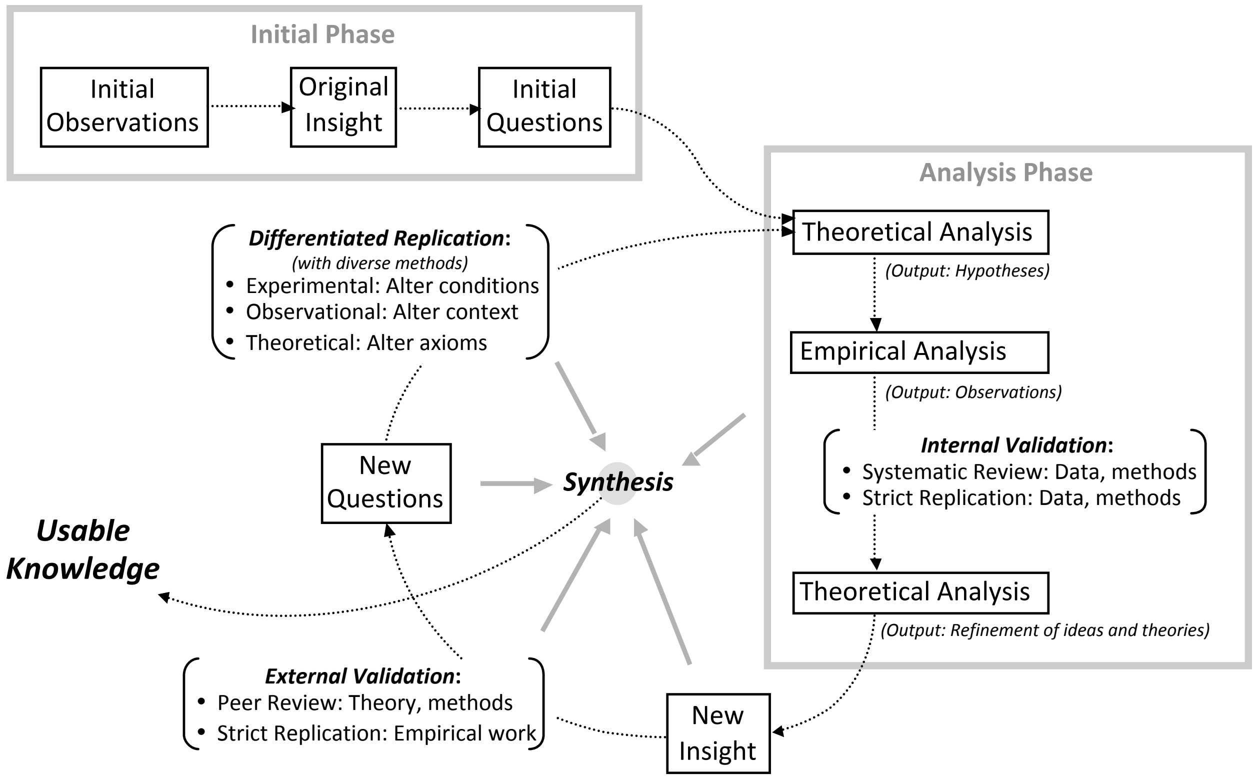

Initial Phase
Initial Observations
Original Insight
Initial Questions
Analysis Phase
Theoretical Analysis
(Output: Hypotheses)
Empirical Analysis
(Output: Observations)
Internal Validation:
• Systematic Review: Data, methods
• Strict Replication: Data, methods
Theoretical Analysis
(Output: Refinement of ideas and theories)
Differentiated Replication:
(with diverse methods)
• Experimental: Alter conditions
• Observational: Alter context
• Theoretical: Alter axioms
New Questions
Synthesis
Usable Knowledge
External Validation:
• Peer Review: Theory, methods
• Strict Replication: Empirical work
New Insight

Figure 6.2: Cycle of Maturing Knowledge (CMK), a unified framework for research methodologies in software engineering.

everywhere; and 4) as we revisit and re-experience the concept in different contexts, we refine our understanding of it. In the spirit of Alexander, these "ah-ha" moments, followed by the iterative process portrayed here, represent a fundamental pattern of learning, which the CMK attempts to model. Therefore, an initial observation (or set of observations), followed by initial insight and initial questions, leads us to a deeper and more methodical analysis.

Within the cycle, each of the *boxed* items represents an element that inevitably occurs, independent of whether it is formally or consciously expressed. For instance, a researcher cannot perform an empirical study without considering at some level the possible outcomes of that study. Accordingly, a researcher cannot help but ponder the implications of new observations. Each of the boxed items, therefore, expresses a feature of the natural human pattern of learning, by which we develop simplified models of complex environments. Conversely, the *bracketed* items represent features that require formalized methods and conscious effort if they are to occur.[3]

Examining the CMK, it becomes clear that strict replication is *one* method for validating empirical observations, and although essential to the cycle, it serves a supporting role (rather than a fundamental or preeminent role) in the process of building knowledge. Conversely, differentiated replication is the primary mechanism driving the knowledge building process, without which the cycle ceases to productively generate new theories and underlying conceptual foundations that may then be formally analyzed.

In producing the CMK, we explicitly incorporate the multi-method research approach [24], which stipulates that differentiated replications must be performed using diverse methods in order to prevent methodology bias [39]. The CMK also expresses key elements of Basili, Shull, and Lanubile's knowledge-building framework, "Families of Experiments" [9], which describes the need to iterate the cycle of analysis numerous times—thus performing

[3]Note that we use the terms *internal* and *external* in this diagram in the same context as Brooks et al. [17], to denote validation that is conducted by the original researchers as opposed to a third party. This usage differs from that of Campbell and Stanley [20], who use the terms to talk about the degree of confidence in cause-effect conclusions, as opposed to the generalizability of results.

a “family” of related differentiated replications—in order to study a particular question or theory.

Basili et al. [9] also describe methods for synthesizing replications. Synthesis methods are represented in the center of the cycle, and incorporate the processes by which usable knowledge is distilled from a “family” of differentiated replications. Juristo and Vegas [48] have contributed significantly to these processes. However, only recently have we begun to recognize the importance of differentiated replication (as opposed to strict replication) in the process of knowledge building. These processes of synthesis are, therefore, still immature and demand further analysis to refine appropriate and effective methods.

Assertion 9. *The Cycle of Maturing Knowledge unifies the current SE research methodologies and represents a SE-specific embodiment of the general knowledge building process.*

6.7 Conclusions

In an attempt to clarify research methodologies in SE, we have synthesized the contributions of numerous scientists and researchers. We are indebted to them for their conceptual, theoretical, and philosophical contributions derived from extensive experience with empirical research.

From this synthesis process, we identify two general principles: 1) In order to discover fundamental patterns on which to build generalizable theories, SE must embrace and account for its true complexity, rather than ignore or abstract it away. 2) In order to avoid premature generalization in the search for fundamental patterns, SE must develop and motivate the use of processes that appropriately and effectively blend theoretical and empirical work. These processes must be founded upon deep synthesis of methodologically diverse studies.

As a reflection of these principles, the CMK identifies three areas for process improvement in SE research: 1) In addition to strict replication, we must explore additional methods for internal and external validation of observations. 2) We need to continue developing methods for conducting differentiated replications, such that results can be synthesized (e.g., improving traceability [48]). 3) We must spend more time synthesizing—that is, iterating the knowledge-building cycle, conducting "families" of differentiated replications [9], and unifying our observations through theory building.

Alas, the search space is vast and sometimes feels daunting. We resonate with the anonymous author who wrote the following inspired summary:

> *We have not succeeded in answering all our problems. The answers we have found only serve to raise a whole set of new questions. In some ways we feel we are as confused as ever, but we believe we are confused on a higher level, and about more important things.*

References

[1] Christopher Alexander. *The Timeless Way of Building*. Oxford University Press, Oxford, NY, USA, 1979.

[2] Christopher Alexander, Sara Ishikawa, Murray Silverstein, Max Jacobson, Ingrid Fiksdahl-King, and Shlomo Angel. *A Pattern Language: Towns, Buildings, Construction*. Oxford University Press, Oxford, NY, USA, 1977.

[3] Matthew Van Antwerp and Greg Madey. Advances in the SourceForge Research Data Archive (SRDA). In *International Workshop on Public Data about Software Development*, September 2008.

[4] Phillip G. Armour. The five orders of ignorance: Viewing software development as knowledge acquisition and ignorance reduction. *Communications of the ACM*, 43(10):17–20, 2000.

[5] Martha Augoustinos and Iain Walker. The construction of stereotypes within social psychology: From social cognition to ideology. *Theory & Psychology*, 8(5):629–652, 1998.

[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 R. Basili. The role of experimentation in software engineering: Past, current, and future. In *International Conference on Software Engineering*, pages 442–449, Washington, DC, USA, 1996. IEEE Computer Society.

[8] Victor R. Basili, Frank E. McGarry, Rose Pajerski, and Marvin V. Zelkowitz. Lessons learned from 25 years of process improvement: The rise and fall of the NASA Software Engineering Laboratory. In *International Conference on Software Engineering*, pages 69–79, Los Alamitos, CA, USA, 2002. IEEE Computer Society.

[9] Victor R. Basili, Forrest Shull, and Filippo Lanubile. Building knowledge through families of experiments. *IEEE Transactions on Software Engineering*, 25(4):456–473, 1999.

[10] 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.

[11] Alessandro Bianchi, Danilo Caivano, Filippo Lanubile, and Giuseppe Visaggio. Evaluating software degradation through entropy. In *International Symposium on Software Metrics*, pages 210–219, Los Alamitos, CA, USA, 2001. IEEE Computer Society.

[12] Christian Bird, Alex Gourley, Prem Devanbu, Anand Swaminathan, and Greta Hsu. Open borders? immigration in open source projects. In *International Workshop on Mining Software Repositories*, Los Alamitos, CA, USA, 2007. IEEE Computer Society.

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

[14] Barry Boehm. Managing software productivity and reuse. *IEEE Computer*, 32(9):111–113, 1999.

[15] Lionel C. Briand, Sandro Morasca, and Victor R. Basili. Defining and validating measures for object-based high-level design. *IEEE Transactions on Software Engineering*, 25(5):722–743, Sep/Oct 1999.

[16] Andrew Brooks, John Daly, James Miller, Marc Roper, and Murray Wood. Replication of experimental results in software engineering. Technical report, Department of Computer Science, University of Strathclyde, Glasgow, UK, 1995.

[17] Andrew Brooks, Marc Roper, Murray Wood, John Daly, and James Miller. Replication's role in software engineering. In Forrest Shull, Janice Singer, and Dag I. K. Sjøberg, editors, *Guide to Advanced Empirical Software Engineering*, pages 365–379. Springer, 2008.

[18] Frederick P. Brooks. No silver bullet: Essence and accidents of software engineering. *IEEE computer*, 20(4):10–19, 1987.

[19] Frederick P. Brooks. *The Mythical Man-Month: Essays on Software Engineering.* Addison-Wesley, Boston, MA, USA, 1995.

[20] Donald T. Campbell and Julian Stanley. *Experimental and Quasi-Experimental Designs for Research.* Houghton Mifflin Co., Boston, MA, USA, 1963.

[21] David N. Card, Frank E. McGarry, and Gerald T. Page. Evaluating software engineering technologies. *IEEE Transactions on Software Engineering*, 13(7):845–851, 1987.

[22] Samuel D. Conte, H. E. Dunsmore, and V. Y. Shen. *Software Engineering Metrics and Models*. Benjamin/Cummings Series in Software Engineering. Benjamin/Cummings Publishing Company, Inc., Menlo Park, CA, USA, 1986.

[23] Bill Curtis. Substantiating programmer variability. *Proceedings of the IEEE*, 69(7):846–846, July 1981.

[24] John W. Daly. *Replication and a Multi-Method Approach to Empirical Software Engineering Research*. PhD thesis, Department of Computer Science, University of Strathclyde, Glasgow, UK, 1996.

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

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

[27] Daniel P. Delorey, Charles D. Knutson, and Mark Davies. Mining programming language vocabularies from source code. In *Psychology of Programming Interest Group Conference*, June 2009.

[28] 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 *International Workshop on Public Data about Software Development*, June 2007.

[29] Daniel P. Delorey, Charles D. Knutson, and Alex MacLean. Studying production phase SourceForge projects: A case study using cvs2mysql and SFRA+. In *International Workshop on Public Data about Software Development*, June 2007.

[30] Tom DeMarco and Timothy Lister. Programmer performance and the effects of the workplace. In *International Conference on Software Engineering*, pages 268–272, Los Alamitos, CA, USA, 1985. IEEE Computer Society.

[31] Mark Dowson. The Ariane 5 software failure. *ACM SIGSOFT Software Engineering Notes*, 22(2):84–84, 1997.

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

[33] Anne Smith Duncan. Software development productivity tools and metrics. In *International Conference on Software Engineering*, pages 41–48, Los Alamitos, CA, USA, 1988. IEEE Computer Society.

[34] Jane E. Dutton. The making of organizational opportunities: An interpretive pathway to organizational change. *Research in Organizational Behavior*, 15:195–226, 1993.

[35] Steve Easterbrook, Janice Singer, Margaret-Anne Storey, and Daniela Damian. Selecting empirical methods for software engineering research. In Forrest Shull, Janice Singer, and Dag I. K. Sjøberg, editors, *Guide to Advanced Empirical Software Engineering*, pages 285–311. Springer, 2008.

[36] 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.

[37] Michael Fischer, Martin Pinzger, and Harald Gall. Populating a release history database from version control and bug tracking systems. In *International Conference on Software Maintenance*, pages 23–32, Washington, DC, USA, 2003. IEEE Computer Society.

[38] Carlo Ghezzi. Reflections on 40+ years of software engineering research and beyond: an insider's view. Keynote address. In *International Conference on Software Engineering*, May 2009.

[39] Saul Greenberg and Bill Buxton. Usability evaluation considered harmful (some of the time). In *Conference on Human Factors in Computing Systems*, pages 111–120, New York, NY, USA, 2008. ACM.

[40] Warren Harrison. An entropy-based measure of software complexity. *IEEE Transactions on Software Engineering*, 18(11):1025–1029, November 1992.

[41] Ahmed E. Hassan. Predicting faults using the complexity of code changes. In *International Conference on Software Engineering*, pages 78–88, New York, NY, USA, 2009. ACM.

[42] Kieran Healy and Alan Schussman. The ecology of open-source software development. Technical report, University of Arizona, Tucson, AZ, USA, 2003.

[43] James Howison and Kevin Crowston. The perils and pitfalls of mining SourceForge. In *International Workshop on Mining Software Repositories*, pages 7–11, 2004.

[44] IEEE Computer Society. IEEE standard for software productivity metrics. *IEEE Std 1045-1992*, 1993.

[45] Capers Jones. Backfiring: Converting lines of code to function points. *IEEE Computer*, 28(11):87–88, 1995.

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

[47] Magne Jørgensen and Dag Sjøberg. Generalization and theory-building in software engineering research. In *International Conference on Empirical Assessment in Software Engineering*, pages 29–36. IEEE Computer Society, 2004.

[48] Natalia Juristo and Sira Vegas. Using differences among replications of software engineering experiments to gain knowledge. In *International Symposium on Empirical Software Engineering and Measurement*, pages 356–366, Washington, DC, USA, 2009. IEEE Computer Society.

[49] Barbara A. Kitchenham. The role of replications in empirical software engineering—a word of warning. *Empirical Software Engineering*, 13(2):219–221, 2008.

[50] Barbara A. Kitchenham, Tore Dybå, and Magne Jørgensen. Evidence-based software engineering. In *International Conference on Software Engineering*, pages 273–281, Washington, DC, USA, 2004. IEEE Computer Society.

[51] Barbara A. Kitchenham, Shari Lawrence Pfleeger, Lesley M. Pickard, Peter W. Jones, David C. Hoaglin, Khaled El Emam, and Jarrett Rosenberg. Preliminary guidelines for empirical research in software engineering. *IEEE Transactions on Software Engineering*, 28(8):721–734, 2002.

[52] Charles D. Knutson. *Pattern Systems and Methodologies for Multiparadigm Analysis and Design*. PhD thesis, Department of Computer Science, Oregon State University, Corvallis, OR, USA, 1998.

[53] Charles D. Knutson, Jonathan L. Krein, Lutz Prechelt, and Natalia Juristo. 1st international workshop on replication in empirical software engineering research (RESER). In *International Conference on Software Engineering*, pages 461–462, New York, NY, USA, May 2010. ACM.

[54] Charles D. Knutson, Jonathan L. Krein, Lutz Prechelt, and Natalia Juristo. Report from the 1st international workshop on replication in empirical software engineering research (RESER 2010). *ACM SIGSOFT Software Engineering Notes*, 35(5):42–44, September 2010.

[55] Jonathan L. Krein and Charles D. Knutson. A case for replication: Synthesizing research methodologies in software engineering. In *International Workshop on Replication in Empirical Software Engineering Research*, May 2010.

[56] 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. In *International Workshop on Public Data about Software Development*, June 2009.

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

[58] Thomas S. Kuhn. *The Structure of Scientific Revolutions*. The University of Chicago Press, Chicago, IL, USA, 3rd edition, 1996.

[59] Nancy G. Leveson. High-pressure steam engines and computer software. In *International Conference on Software Engineering*, pages 2–14, New York, NY, USA, 1992. ACM.

[60] Nancy G. Leveson and Clark S. Turner. An investigation of the Therac-25 accidents. *IEEE Computer*, 26(7):18–41, 1993.

[61] Jonathan Lung, Jorge Aranda, Steve Easterbrook, and Gregory Wilson. On the difficulty of replicating human subjects studies in software engineering. In *International Conference on Software Engineering*, pages 191–200, New York, NY, USA, 2008. ACM.

[62] 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 *International Workshop on Replication in Empirical Software Engineering Research*, May 2010.

[63] Alexander C. MacLean, Landon J. Pratt, Jonathan L. Krein, and Charles D. Knutson. Trends that affect temporal analysis using SourceForge data. In *International Workshop on Public Data about Software Development*, June 2010.

[64] Leo Marx and Merritt R. Smith. *Does Technology Drive History?: The Dilemma of Technological Determinism.* MIT Press, Cambridge, MA, USA, 1994.

[65] 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.

[66] Steve McConnell. 10 most powerful ideas in software engineering. Keynote address. In *International Conference on Software Engineering*, May 2009.

[67] Pam McDonald, Dan Strickland, and Charles Wildman. Estimating the effective size of autogenerated code in a large software project. In *International Forum on COCOMO and Software Cost Modeling*, October 2002.

[68] 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.

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

[70] Peter Naur. Programming languages, natural languages, and mathematics. *Communications of the ACM*, 18(12):676–683, 1975.

[71] Peter Naur. *Programming Languages are not Languages: Why 'Programming Language' is a Misleading Designation.* Addison-Wesley, Reading, MA, USA, 1991.

[72] Edward Nelson. Management handbook for the estimation of computer programming costs. Technical report, Systems Development Corporation, Santa Monica, CA, USA, 1966.

[73] Wanda J. Orlikowski and C. Suzanne Iacono. The truth is not out there: An enacted view of the "digital economy". In Erik Brynjolfsson and Brian Kahin, editors, *Understanding the Digital Economy: Data, Tools, and Research*, pages 352–380. MIT Press, Cambridge, MA, USA, 2000.

[74] Dewayne E. Perry, Adam A. Porter, and Lawrence G. Votta. Empirical studies of software engineering: A roadmap. In *Future of Software Engineering Symposium*, pages 345–355, New York, NY, USA, 2000. ACM.

[75] Karl R. Popper. *The Logic of Scientific Discovery.* Reprint, Routledge, New York, NY, USA, [1959] 2000.

[76] Lutz Prechelt. The 28:1 Grant/Sackman legend is misleading, or: How large is interpersonal variation really? Technical report, Universität Karlsruhe, Fakultät für Informatik, Karlsruhe, Germany, 1999.

[77] 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, CA, USA, 2nd edition, 2002.

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

[79] William D. Ruckelshaus. Risk, science, and democracy. In Theodore S. Glickman and Michael Gough, editors, *Readings in Risk*. Resources for the Future, Washington, DC, USA, 1990.

[80] Samuel H. Scudder. Take this fish and look at it. In *The Prentice Hall Guide for College Writers*, pages 53–56. Allyn & Bacon, 9th edition, 2010.

[81] Claude E. Shannon. A mathematical theory of communication. *The Bell System Technical Journal*, 27:379–423, 623–656, Jul/Oct 1948.

[82] Forrest Shull, Victor Basili, Jeffrey Carver, José Maldonado, Guilherme Travassos, Manoel Mendonça, and Sandra Fabbri. Replicating software engineering experiments: Addressing the tacit knowledge problem. In *International Symposium on Empirical Software Engineering*, pages 7–16, Los Alamitos, CA, USA, 2002. IEEE Computer Society.

[83] Forrest Shull, Jeffrey Carver, Sira Vegas, and Natalia Juristo. The role of replications in empirical software engineering. *Empirical Software Engineering*, 13(2):211–218, 2008.

[84] Forrest Shull and Raimund L. Feldmann. Building theories from multiple evidence sources. In Forrest Shull, Janice Singer, and Dag I. K. Sjøberg, editors, *Guide to Advanced Empirical Software Engineering*, pages 337–364. Springer, 2008.

[85] Dag I. K. Sjøberg, Tore Dybå, Bente C. D. Anda, and Jo E. Hannay. Building theories in software engineering. In Forrest Shull, Janice Singer, and Dag I. K. Sjøberg, editors, *Guide to Advanced Empirical Software Engineering*, pages 312–336. Springer, 2008.

[86] Dag I. K. Sjøberg, Tore Dybå, and Magne Jørgensen. The future of empirical methods in software engineering research. In *Future of Software Engineering Symposium*, pages 358–378, Washington, DC, USA, 2007. IEEE Computer Society.

[87] Steve Smith, Amelia Hadfield, and Timothy Dunne. *Foreign Policy: Theories, Actors, Cases*. Oxford University Press, Oxford, NY, USA, 2008.

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

[89] Quinn C. Taylor, James E. Stevenson, Daniel P. Delorey, and Charles D. Knutson. Author entropy: A metric for characterization of software authorship patterns. In *International Workshop on Public Data about Software Development*, September 2008.

[90] Walter F. Tichy, Paul Lukowicz, Lutz Prechelt, and Ernst A. Heinz. Experimental evaluation in computer science: A quantitative study. *Journal of Systems and Software*, 28(1):9–18, 1995.

[91] 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.

[92] Shinji Uchida, 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.

[93] Brian Wynne. Episode 10. In Paul Kennedy, host, David Cayley, producer, *CBC IDEAS: How to Think About Science*, radio broadcast. Canadian Broadcasting Corporation, Toronto, ON, Canada, 2008.

[94] Jin Xu, Yongqin Gao, Scott Christley, and Gregory Madey. A topological analysis of the open souce software development community. In *Hawaii International Conference on System Sciences*, Washington, DC, USA, 2005. IEEE Computer Society.

[95] 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.