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

Jonathan L. Krein, Alexander C. MacLean, Charles D. Knutson
SEQuOIA Lab, Department of Computer Science, Brigham Young University
jonathankrein@byu.net, amaclean@byu.net, knutson@cs.byu.edu

Daniel P. Delorey
Google, Inc.
dalorey@google.com

Dennis L. Egget
Department of Statistics,
Brigham Young University
thecgg@stat.byu.edu

ABSTRACT
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.

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

Baldo et al. define language as a “rule-based, symbolic representation system” that “allows us to not simply represent concepts, but more importantly for problem solving, facilitates our ability to manipulate those concepts and generate novel solutions” [2]. Although their study focused on the relationship between natural language and problem solving, their concept of language is highly representative of languages used in programming activities. Other research in the area of linguistics examines the differences between mono-, bi-, and multilingual speakers. One particular study, focusing on the differences between mono- and bilingual children, found specific differences in the subjects’ abilities to solve problems [3]. 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 2 and 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 4 presents the thesis of the paper, and Sections 5 and 6 describe, justify, and validate the data and analysis techniques. We then present in Section 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.

2. PRODUCTIVITY

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

In the 1960’s, Edward Nelson performed one of the earliest studies to identify programmer productivity factors [22]. Nelson found that programmer productivity correlates with at least 15 factors. More recently (2000), Capers Jones identified approximately 250 factors that he claims influence programmer performance [19]. Summarizing this research, Endres and Rombach state that reducing productivity to “ten or 15 parameters is certainly a major simplification” [15]. With so many contributing factors to measure, it is not surprising that numerous productivity metrics have been proposed in the literature.

Nevertheless, all reasonable productivity metrics intercorrelate to some degree, and all productivity metrics suffer from threats to validity—the significance of those threats depends on the circumstances in which the metrics are applied. The researcher, therefore, must weigh the trade-offs and select a suitable metric based on the available data and the context of the study. For a discussion of the trade-offs inherent in various common productivity metrics, as well as an overview of the primary threats to the validity of those metrics, we refer the reader to work by Conte, Dunsmore, and Shen [8] and to work by Endres and Rombach [15]. The most common software productivity metrics include function points and lines of code (LOC).

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

In the literature, the list of studies that rely on LOC and time primitives to estimate productivity is lengthy. Studies that use these primitives (e.g., [27] [5] [14] [7]) 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 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 5.2.2). 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 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. 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 [20]. 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 [26].

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 [16] [25]. 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 [17] and [4], but none of them apply entropy to language fragmentation.

3.2 Calculation

The general formula for calculating the entropy of a system in information theory is shown in Equation 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 $i$.

$$E(S) = - \sum_{i=1}^{c} (p_i \cdot \log_2 p_i)$$  \hspace{1cm} (1)$$

To apply this general entropy formula to language fragmentation, we specifically define the variables in Equation 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 1).

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 1, we arrive at a discrete function for the maximum possible entropy for a given number of languages (see Equation 2).

$$E_{max} = \log_2(c)$$  \hspace{1cm} (2)$$

Notice in Equation 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 and Table 1).

$$\lim_{c \to \infty} E_{max}(c + 1) - E_{max}(c) = 0$$ \hspace{1cm} (3)$$

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

However, considering the case of a person who already knows multiple languages from the same paradigm (say imperative), it is unclear whether the addition of a new language from a different paradigm (say object-oriented) would, in reality, impact the developer less than the addition of the previous language from the familiar paradigm. More generally, we suspect that the addition of a language from an unfamiliar paradigm would result in a more dramatic impact than would the addition of a language from a familiar paradigm (see Section 10.3).

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

\(^1\)The log operation is undefined at zero; thus, languages with a $p_i = 0$ must be excluded from the calculation. As a result, for all values $c > 1$ the minimum language entropy of 0 occurs in the limit as some $p_i$ approaches 1.
The project was under active development as of August 2006; source; 2) the project utilized CVS for revision control; 3) match the following four criteria: 1) the project is open

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.

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 [12].

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 [12].

A study by Delorey, Knutson, and Chun [10] 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 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.

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.

5.2.1 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.

<table>
<thead>
<tr>
<th># of Languages</th>
<th>Min. Entropy</th>
<th>Max. Entropy</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0.00</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>1.00</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>1.59</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>2.00</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>2.32</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>49</td>
<td>0</td>
<td>5.62</td>
</tr>
<tr>
<td>50</td>
<td>0</td>
<td>5.64</td>
</tr>
</tbody>
</table>

Table 1: Entropy ranges for sample language cases

<table>
<thead>
<tr>
<th>Project Rank</th>
<th>Author Rank</th>
<th>File Rank</th>
<th>Revision Rank</th>
<th>LOC Rank</th>
<th>Final Rank</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>Java</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>C++</td>
<td>4</td>
<td>3</td>
<td>4</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>PHP</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>Python</td>
<td>7</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>Perl</td>
<td>3</td>
<td>5</td>
<td>9</td>
<td>9</td>
<td>6</td>
</tr>
<tr>
<td>JavaScript</td>
<td>6</td>
<td>6</td>
<td>8</td>
<td>8</td>
<td>10</td>
</tr>
<tr>
<td>C#</td>
<td>9</td>
<td>9</td>
<td>7</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>Pascal</td>
<td>8</td>
<td>10</td>
<td>8</td>
<td>7</td>
<td>9</td>
</tr>
<tr>
<td>Tcl</td>
<td>11</td>
<td>8</td>
<td>10</td>
<td>10</td>
<td>9</td>
</tr>
</tbody>
</table>

Table 2: Top ten programming languages by popularity rankings (account for 99% of the lines of code in the data set)

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 [12].
5.2.2 Filtering the Data
To help account for multi-month code submissions, as well as the six factors identified by Delorey et al. [10]—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. [10], 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.

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 4 and 5, respectively). We conclude this section by justifying and validating the selected model.

6.1 Transforming the Data
Figure 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 2(b) that outliers are minimized, the distribution is approximately normal, and the range is controlled.

6.2 Selecting a Statistical Model
Figure 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) is masked by crowding, which causes points to be plotted over other points. In total, nearly 5,000 points are plotted, of which approximately 48% lie on the y-axis at the entropy value of zero. Thus, nearly half the data consists of months in which developers submitted code in only one language. The distribution of the data points with respect to language entropy is consistent with the findings of Delorey, Knutson, and Giraud-Carrier who, for the same SourceForge data set, report that approximately 70% of developers write in a single language per year [11].

The relative density of the data is much easier to see in Figure 4. Density maps 4(a) and 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 4(c) and 4(d) increase the contrast by calculating the densities for only the data with entropy values greater than zero.

Since this study intends to show a significant relationship between language entropy and lines contributed, we must demonstrate both a significant correlation between the two metrics and a reasonable variance in the data. The data plot and density maps, however, show a large spread in the data, indicating considerable variance. For the non-zero-entropy data (not on the y-axis), there does not appear to be a sig-
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 [24]. However, to be conservative we assume that serial correlation exists in the data and adjust for it in our analysis.

6.4 Banding in the Data
The scatter plot in Figure 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 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 1).

Figure 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.

6.4.1 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, 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 5(a) that the restricted area at the origin and/or the slope of the bands may cause the slope of a regression line to be inaccurately negative.

One method for validating the regression model is to test it on data for which the correlation between language entropy and lines contributed is known. Therefore, we produce an artificial data sample for the two-language case such that no correlation exists between language entropy and lines contributed. We generate our artificial sample by replacing all contributions in which all but two lines were written in the same language 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.
Figure 4: Relative density maps of ln(Lines Contributed) vs. Language 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 $E(S) > 0$

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 5(b)). Logging the dependent variable does not compromise the analysis because the compression ratio of the log transformation increases exponentially as its argument increases linearly. In effect, the transformation’s amplification of the low-range data is counteracted by the way it compresses the high-range data more significantly, causing the analysis to appropriately place greater weight on the high range.

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

Although the mean for the zero-entropy data and the slope of the regression line for the non-zero-entropy data are not affected by the data mapping, the intercept of the regression line is affected. The analysis of the artificial, non-correlated data yields an intercept of 3,311 LOC—809 lines above the median of the data range (2,500 LOC). The positive shift in the intercept of the regression line is another artifact of the interplay between the metrics, which results in a mapping of
the data into a space with a density gradient that increases radially from the origin (see Figure 5(b)). Thus even before taking the natural log of the dependent variable, the high-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.

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). This pattern is a consequence of the distribution of the data points. Delorey, Knutson, and Giraud-Carrier found in their analysis of the SourceForge data set that only 10% of developers use more than two languages per year [11]. 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 4).

7. RESULTS
Table 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 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 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 data. However, as discussed in Section 6.4.1, a positive difference between the intercept of the regression line and the estimate of the mean at entropy zero may be nothing more than an artifact of the metrics. Adding the first two parameter estimates gives the estimate for the intercept of the NZG regression line (6.2661). The third parameter, nonZeroEntropyGroupSlope, represents the slope of the NZG regression line, which is negatively correlated with language entropy.

The magnitudes of these parameter estimates make more sense on the original scale. However, the back-transformed estimates must be reinterpreted because the analysis is performed on log-transformed data. Specifically, the ZG mean and the intercept of the NZG regression line both represent medians on the original scale. Further, the slope of the NZG regression line becomes a multiplicative factor, which means that an increase in language entropy results in a multiplicative decrease in lines contributed. Equations 4 and 5 show
the back-transformed model, and Figure 6 shows the model graphed on the normal scale.

$$ZG_{median} = e^{4.0678}$$
$$= 58.4$$

$$NZG = e^{(4.0678+2.1983)x -0.5072x}$$
$$= 526.4(e^{-0.5x})$$

Figure 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 6.4.1), 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).

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.

9. LIMITATIONS

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

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 10.1).
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 5.2.1 and 5.2.2). 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.

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.

9.4 Marginally Active Developers
Developers who make only small contributions per month may bias the analysis results. First, these marginally active developers are probably less likely to write in multiple languages during a given month. In this case, filtering them out could reduce the disparity between the zero- and non-zero-entropy groups (especially considering the power law trends found by Healy and Schussman in SourceForge data [18]). 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.

10. FUTURE WORK
In this section we outline avenues for future research.

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 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.

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?

- 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?

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.

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?

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 [6].

11. ACKNOWLEDGEMENTS
We are grateful to reviewers and attendees of the 4th Workshop on Public Data about Software Development (WoP-DaSD 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.

12. REFERENCES

3Lest 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.


