Do Programming Languages Affect Productivity? A Case Study Using Data from Open Source Projects

Daniel P. Delorey
Brigham Young University
Provo, UT
pierce@cs.byu.edu

Charles D. Knutson
Brigham Young University
Provo, UT
knutson@cs.byu.edu

Scott Chun
Brigham Young University
Provo, UT
chun@cs.byu.edu

Abstract

Brooks and others long ago suggested that on average computer programmers write the same number of lines of code in a given amount of time regardless of the programming language used. We examine data collected from the CVS repositories of 9,999 open source projects hosted on SourceForge.net to test this assumption for 10 of the most popular programming languages in use in the open source community. We find that for 24 of the 45 pairwise comparisons, the programming language is a significant factor in determining the rate at which source code is written, even after accounting for variations between programmers and projects.

1 Introduction

Brooks is generally credited with the assertion that annual lines-of-code programmer productivity is constant, independent of programming language. In making this assertion, Brooks cites multiple authors including [7] and [8]. Brooks states, "Productivity seems constant in terms of elementary statements, a conclusion that is reasonable in terms of the thought a statement requires and the errors it may include." [1] (p. 94) This statement, as well as the works it cites, however, appears to be based primarily on anecdotal evidence. We test this assertion across ten programming languages using data from open source software projects.

2 Related Work

Various studies of productivity in software development have been reported, including [5, 4, 6, 3].

Empirical studies of programmer productivity differ in the productivity measures used, the types and quantities of data used, the explanatory factors considered, the goals of the study, and the conclusions reached.

The most common productivity metrics are lines of code per unit time [5] and function points per unit time [4, 6, 3]. While compelling arguments are made in the literature for both of these metrics, we use lines of code both because the assertion we are testing was stated in terms of lines of code.

Studies of software development productivity tend to rely on observational data collected from commercial projects. Maxwell et al. use data collected from 99 projects from 37 companies in eight European countries [5] and data gathered from 206 projects from 26 companies in Finland [4]. Premraj et al. use an updated version of the same data set with over 600 projects [6]. Liebchen et al. use a data set representing more than 25,000 projects from a single company [3]. Our data set was collected from the CVS repositories of 9,999 open source projects hosted on SourceForge.

The data sets used in these studies were each compiled manually with some level of subjectivity and transformation. Given this level of human involvement, the factors they consider are at a high level of abstraction. For example, the data set in [5] contains among its variables seven COCOMO factors, including required reliability, execution time constraints, and main storage constraints, each with discrete ordinal values between 1 and 6. Our data set contains only those features that can be calculated from the data in a CVS repository. As such, our data is limited conceptually but has the advantages of being concrete, objective, and simple to gather.

In each of the papers cited, the stated goal of the study was to identify the major factors influencing programmer productivity. The models developed in these studies were intended to be either predictive, explanatory, or both. Our goal is not to construct a predictive or explanatory model. Rather, we seek only to develop a model that sufficiently accounts for the variation in our data so that we may test the significance of the

First International Workshop on Emerging Trends in FLOSS Research and Development (FLOSS'07)
0-7695-2961-5/07 $20.00 © 2007 IEEE

Table 1. Top ten programming languages by popularity rankings

	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

estimated effect of programming language.

3 Data Collection

The data we use in our analysis comes from the CVS repositories of open source projects hosted on SourceForge. The tools we developed and methods we employed in collecting the data are described in [2].

As CVS manages individual changes (called *revisions*) it records the author of the change, the date and time the change happened, the number of lines that were added to and removed from the file, and a mandatory free-form message supplied by the author. These minimal data can be combined to produce a rich set of values describing the environment in which the change was made.

We collected data from the CVS repositories of 9,999 projects hosted on SourceForge. Our population for the data collection was the set of projects that met the following criteria: 1) the project's development stage is set as Production/Stable or Maintenance; 2) the project is active; 3) the project uses CVS; 4) the project is open source.

We gathered the entire history for each of the 9,999 CVS repositories and stored the resulting data in a MySQL relational database using a tool we developed called `cvs2mysql` [2]. The resulting raw data contains records for 7,244,201 files and 26,559,460 changes to those files made by 23,838 developers.

3.1 Data Preparation

Of the more than 19,000 different file extensions represented in the SourceForge database, we identified 107 unique programming language extensions. In order to limit the scope of our study to the languages that are most widely used, we produced an ordered list of the most popular programming languages represented in the database. Popularity is defined here in terms of: 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. We ranked each language using these five metrics and calculated the average ranking for each language. We then ranked the languages by their average rankings to determine an overall ranking. We chose to focus on the top 10 programming languages which are listed along with their rankings in Table 1. These 10 languages are used in 89% of all projects, by 92% of all authors, and account for 98% of the files, 98% of the revisions, 99% of the lines of code in our data set. The next three most popular languages are Prolog, Lisp, and Scheme, none of which can be easily compared to imperative and object-oriented languages on a line by line basis given the differences in programming paradigm.

We compare annual productions per programmer per language in an effort to limit the impact of normal variations in the amount of time individual programmers commit to development over smaller time periods. Data collection was limited to the time period from January 1, 2000 to December 31, 2005.

Our model of aggregating the lines written across authors, programming languages, and years assumes that every line committed to CVS by an author was written by that author during the year in which it was committed. However, we identified six ways in which this assumption can be violated:

- Migration – An existing CVS repository created by multiple authors and/or over multiple years is migrated to SourceForge by a single author.
- Dead File Restoration – When a dead file is restored in CVS, the contents are not differenced against the pre-removal version.
- Multi-Project Files – Authors may contribute the same file to multiple projects.
- Gatekeepers – Gatekeepers receive credit for all the lines they commit even if they were not the author.
- Batch Commits – An author may work for more than a year before committing the changes.
- Automatic Code Generation – The tools an author uses to program may automatically generate lines of code which the author then commits to CVS.

While the data collected by CVS does not allow us to definitively identify all cases that violate our assumptions, we have taken steps to exclude as many offending cases as possible while sacrificing as few of the cases that do not violate our assumptions as is reasonable. To remove the migration cases, we excluded initial revisions for all files in our data set. To remove the dead file restoration cases, we excluded all

Table 2. Potential explanatory factors considered

Language Related Factors Per Year	**Author Related Factors Per Year**
For the Current Year	**For the Current Year**
Months since first recorded use	Months since first contribution
Active projects using this language	Active projects with contributions
Active authors using this language	Number of programming languages used
Current files written in this language	Current files edited
Total number of lines written in this language	Total number of lines written
Aggregated Over Prior Years	**Aggregated Over Prior Years**
Total projects having used this language	Total projects with contributions
Total authors having used this language	Total number of programming languages used
Total files written in this language	Total files edited by this author
Total number of lines written in this language	Total number of lines written by this author
Language Specific Author Related Factors Per Year	
For the Current Year	**Aggregated Over Prior Years**
Months since first contribution	Total number of lines written
Active projects with contributions	Total projects with contributions
Current files edited	Total files edited by this author
Temporal Factor Calendar Year	

revisions that followed a "dead" revision. After removing these, however, significant unrealistic outliers remained in our data set. To remove these outliers, we limited our population to those authors who had written fewer than 80,000 lines of source code in a single year. Since we believe that those authors who wrote more than 80,000 lines in a single year are exhibiting one of the non-population behaviors described above, we also exclude from our analysis the projects to which they contributed.

After limiting target programming languages and removing observations deemed to be outside our population, our target data contains records of 673,528 files, 4,198,724 revisions, and 16,197 authors. These data are aggregated across author, programming language, and year into 34,566 observations in our final data set.

4 Data Analysis

The goal of our data analysis is to determine whether there is evidence in the data we have collected that programming languages affect annual programmer productivity. Our dependant variable in this analysis is the lines of code committed to the CVS repositories of selected SourceForge projects by an individual author in a single year. Our independent variable is the programming language being used. We test all pairwise differences between the languages, adjusting our confidence intervals using the Tukey-Kramer Honest Significant Difference for multiple comparisons.

Clearly there are factors other than programming language that affect programmer productivity. Before testing the significance of the programming language effect, we must account for the effects of these confounding variables. We do this by including the confounding factors in a multiple linear regression analysis along with the independent variables so that their effects can be separated. The potential confounding factors we consider in this analysis are listed in Table 2. It is important to note that our goal is only to separate confounding effects before testing our independent variable. Our model is not intended to be predictive or explanatory. Therefore, we do not report the coefficients or the p-values of the confounding factors.

We develop our model by first excluding the programming language and considering only the confounding factors as independent variables. We systematically remove independent variables until we achieve the simplest model that still explains a significant portion of the variation in our data. To this model we then add the programming language factor and test its significance. The procedure for reducing the model is explained below.

We begin by removing independent variables that are highly correlated. Using correlated independent variables in a multiple regression leads to a condition known as multicolinearity which can affect the precision of estimates in unexpected ways. The Variance Inflation Factor (VIF) is a measure of multicolinearity. A VIF value grater than 10 is considered large. Using multicollinearity analysis we remove five of the independent variables. These variables along with their VIF values are listed in Table 3.

We next remove independent variables that have no explanatory power. To be useful as an independent variable in a multiple linear regression, a variable must have a linear relationship with the dependent variable.

Table 3. Explanatory factors excluded from our analysis

Factors Excluded Due to High Variance Inflation Factors (VIF Value)
Total authors having used the programming language in prior years (1860)
Total authors using the programming language in the current year (258)
Total projects having used the programming language in prior years (68)
Files written in the programming language in the current year (51)
Active projects using the programming language in current years (12)
Factors Excluded Due to Low Correlation with the Dependant Variable (Correlation)
Months since the first recorded use of the programming language (0.0071)
Calendar Year (0.0093)
Factors Excluded Due to Practically Insignificant Coefficients (Coefficient)
Total number of lines written in the language during the current year (0.0000)
Total number of lines written in the language during prior years (0.0001)
Factors Removed During Variable Selection Using the Cp Statistic
Total number of languages used by the author during prior years
Total number of files written in the language during prior years

Correlation is a measure of linear relationship. Using the correlation between each independent variable and the dependant variable methods we are able to remove two of the independent variables. These variables along with their correlation coefficients are listed in Table 3.

Fitting a regression on the remaining variables we find that two of the variables have an estimate coefficient equal to or near zero. These coefficients are not statistically significant, but more importantly, they are not practically significant either, so they are removed. These variables along with their estimated coefficients are listed in Table 3.

Finally, the last step in reducing our model is to fit regressions using all possible subsets of the remaining variables and pick the model that best satisfies a model-fitting criterion. The model fitting criterion we use is the Cp statistic. The Cp statistic focuses directly on the trade-off between bias due to excluding important independent variables and extra variance due to the inclusion of too many variables. Using Cp selection on the remaining 16 independent variables, we find the model with the lowest Cp statistic in which all independent variables are significant contains 14 independent variables. The two independent variables excluded from this model are listed in Table 3.

Our final model contains 14 independent variables. Again, the goal of our analysis is not to create a predictive or an explanatory model but rather to control as much of the variation in the data as possible before testing the significance of the effect of programming language on average annual programmer productivity. Therefore, we do not explicitly present the independent variables included in our model to prevent the casual reader from interpreting our model as explanatory or predictive. For the curious reader, the independent variables included in our model can be determined using Table 2 and Table 3. The R^2 for our model is 0.80 meaning that it explains 80% of the variation in our data. All the independent variables are statistically significant at $p < 0.05$. The model is significant at $p < 0.0001$.

5 Results

To test the assertion that programmer productivity is constant in terms of lines of code per year regardless of the programming language being used, we fit a model consisting of the 14 independent variables selected in Section 4 to adjust for variation in programmer ability and programming language use. To this model, we add indicator variables for the programming languages we are considering. By running the analysis nine times and using a different language as the reference each time, we are able to determine the estimated differences between the languages and the standard errors for each of those estimates which we then use to test the significance of the differences.

The null hypothesis for our tests is that there will be no difference in estimated average annual productions per programmer for any of the languages. However, we find evidence in the data to reject the null hypothesis for 24 of the 45 pair-wise comparisons. The p-values for the comparisons, adjusted using the Tukey-Kramer Honest Significant Difference for multiple comparisons are listed in Table 4. The shaded cells are the comparisons for which we reject the null hypothesis with 95% confidence or greater. To clarify the magnitudes of the differences, Figure 1 shows the estimated average annual productions for each language.

Using Table 4 and Figure 1 together we can observe groupings in the languages. Python, which sits near the

Table 4. Pair-wise language comparisons

	JavaScript	Perl	Tcl	Python	PHP	Java	C	C++	C#
Perl	0.46								
Tcl	0.60	1.00							
Python	0.00	0.00	0.76						
PHP	0.00	0.00	0.08	0.72					
Java	0.00	0.00	0.02	0.18	1.00				
C	0.00	0.00	0.00	0.01	0.53	1.00			
C++	0.00	0.00	0.00	0.00	0.01	0.07	0.59		
C#	0.00	0.00	0.00	0.02	0.26	0.50	0.83	1.00	
Pascal	0.00	0.00	0.00	0.00	0.10	0.26	0.60	0.99	1.00

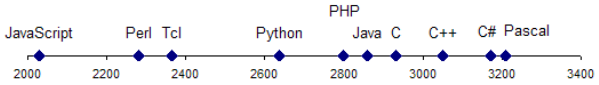

Figure 1. Estimated Average Productions

middle of the range of estimated annual productions, for example, follows a different paradigm from the languages on each end of the range (JavaScript and Perl on the left and C, C++, C#, and Pascal on the right), but it is not significantly different from the other languages near the middle (Tcl, PHP, and Java). Further analysis may reveal that programming language paradigm influences programmer productivity.

6 Conclusions

We find significant evidence in our data that, even after accounting for variations in programmers and environments, programming languages are associated with significant differences in annual programmer productivity. The reader must be careful, however, not to infer a cause-and-effect relationship based solely on this study. Our analysis relies on observational data gathered from SourceForge.net CVS repositories. This is a strength in that the data represent an unaltered software development environment. However, it does limit the inferences we can make both in terms of cause-and-effect and generalization.

Nevertheless, the results of this study suggest a number of interesting avenues for future research. For example, there is a general progression in Figure 1 from newer, higher-level interpreted languages to older, compiled languages. This progression may imply a relationship between the level of abstraction of a language and the speed at which developers can write source code in that language. Brooks supported the assumption of constant productivity as "reasonable in terms of the thought a statement requires and the errors it may include." However, it is quite possible that today's higher-level languages require more thought per line or allow more errors per line than their predecessors. More research is needed to better understand the trade-offs between the power provided by languages with higher levels of abstraction and the cognitive load placed on their users.

We expect that this model of using large-scale, longitudinal studies of Open Source projects to empirically test long-held assumptions in software engineering research will become more prevalent as the tools and methods for collecting and analyzing data from software repositories mature. Such studies are necessary in order to build a more firm foundation for understanding the similarities and differences between Open Source and other software development models.

References

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

[2] D. Delorey, C. Knutson, and A. MacLean. A comprehensive evaluation of production phase sourceforge projects: A case study using cvs2mysql and the sourceforge research archive. *Manuscript Under Review*, 2007.

[3] G. A. Liebchen and M. Shepperd. Software productivity analysis of a large data set and issues of confidentiality and data quality. In *Proceedings of the 11th IEEE International Software Metrics Symposium (METRICS 2005)*, 2005.

[4] K. D. Maxwell and P. Forselius. Benchmarking software development productivity. *IEEE Software*, pages 80–88, January 2000.

[5] K. D. Maxwell, L. V. Wassenhove, and S. Dutta. Software development productivity of european space, military, and industrial applications. *IEEE Transactions on Software Engineering*, 22(10):706–718, October 1996.

[6] R. Premraj, M. Shepperd, B. Kitchenham, and P. Forselius. An empirical analysis of software productivity over time. In *Proceedings of the 11th IEEE International Software Metrics Symposium (METRICS 2005)*, 2005.

[7] W. M. Taliaffero. Modularity. the key to system growth potential. *IEEE Software*, 1(3):245–257, July 1971.

[8] R. W. Wolverton. The cost of developing large-scale software. *IEEE Transactions on Computers*, C-23(6):615–636, June 1974.

0-7695-2961-5/07 $20.00