Assessing Programming Language Impact on Development and Maintenance: A Study on C and C++

Pamela Bhattacharya Iulian Neamtiu
Department of Computer Science and Engineering
University of California, Riverside, CA, USA
{pamelab,neamtiu}@cs.ucr.edu

ABSTRACT

Billions of dollars are spent every year for building and maintaining software. To reduce these costs we must identify the key factors that lead to better software and more productive development. One such key factor, and the focus of our paper, is the choice of programming language. Existing studies that analyze the impact of choice of programming language suffer from several deficiencies with respect to methodology and the applications they consider. For example, they consider applications built by different teams in different languages, hence fail to control for developer competence, or they consider small-sized, infrequently-used, short-lived projects. We propose a novel methodology which controls for development process and developer competence, and quantifies how the choice of programming language impacts software quality and developer productivity. We conduct a study and statistical analysis on a set of long-lived, widely-used, open source projects—Firefox, Blender, VLC, and MySQL. The key novelties of our study are: (1) we only consider projects which have considerable portions of development in two languages, C and C++, and (2) a majority of developers in these projects contribute to both C and C++ code bases. We found that using C++ instead of C results in improved software quality and reduced maintenance effort, and that code bases are shifting from C to C++. Our methodology lays a solid foundation for future studies on comparative advantages of particular programming languages.

Categories and Subject Descriptors

D.2.8 [**Software Engineering**]: Metrics—*complexity measures*; D.2.9 [**Software Engineering**]: Management—*productivity*

General Terms

Languages, Measurement

Keywords

Software quality; developer productivity; software evolution; high-level languages; empirical studies

ICSE '11, May 21–28, 2011, Honolulu, Hawaii, USA

1. INTRODUCTION

Building and maintaining software is expensive. Studies indicate that maintenance costs are at least 50%, and sometimes more than 90%, of the total costs associated with a software product [23, 38]. A NIST survey estimated that the annual cost of software bugs is about $59.5 billion [33]. These findings indicate that there is a pressing need in understanding factors associated with building and maintaining software. At the same time, we are currently witnessing a shift in the language choice for new applications: with the advent of Web 2.0, dynamic, high-level languages are gaining more and more traction [10, 40]; these languages raise the level of abstraction, promising faster development of higher-quality software. However, the lack of static checking and the lack of mature analysis and verification tools makes software written in these languages potentially more prone to error and harder to maintain, so we need a way to quantitatively assess whether they indeed improve development and maintenance.

To that end, in this paper we present a methodology for assessing the impact of programming language on development and maintenance, a long-standing challenge [27]. We first introduce an approach for attributing software quality and ease of maintenance to a particular programing language, then exemplify the approach by comparing C and C++. C++ was designed extending C to include features—object-oriented constructs, overloading, polymorphism, exception handling, stronger typing—aimed at faster construction of less error-prone software. To understand whether using C++ instead of C leads to better, easier to maintain software, we answer several questions directly related to software construction and maintenance: Are programs written in C++ easier to understand and maintain than programs written in C? Are C++ programs less prone to bugs than C programs? Are seasoned developers, with equal expertise in C and C++, more productive in C++ than in C? Are code bases shifting from C to C++?

We answer these questions via an empirical study; we are now in a good position to conduct such a study because both C and C++ are mature, and have been used in large projects for a time long enough to study the effects of using one language versus the other.

Prior efforts on analyzing the impact of choice of programming language suffer from one or more deficiencies with respect to the applications they consider and the manner they conduct their studies: (1) they analyze applications written in a combination of two languages but these applications are *small-sized* and have *short lifespans*, or (2) they consider software built entirely using a *single* language, rather than performing a cross-language evaluation, or (3) they examine applications that are written in *different languages* by *different teams*. Using such methodologies often results in analyses which cannot be generalized to large real-world applications. We aimed to address all these shortcomings.

First, we consider four large, long-lived, open source applications: Mozilla Firefox, VLC Media Player, Blender Animation Software and MySQL; our analysis covers 216 official releases and a combined 40 years of evolution. All these applications are mature, stable, have large code bases in both C and C++, and have large user bases; their long histories help us understand issues that appear in the evolution of multi-developer widely-used software.

Second, we ensure the uniformity of software development process when comparing C and C++ code. Prior work has compared languages by considering applications written exclusively in a single language, e.g., by implementing the same small task in C, C++, Fortran, or Visual Basic [1, 34, 18, 25, 19]. We only studied projects that contain *both* C and C++ code, to guarantee uniformity in the development process of the application.

Third, we effectively control for developer competence to ensure that changes to software facets, e.g., quality, can be attributed to the underlying programming language. We use a statistical analysis of committer distribution to show that the majority of developers contribute to both C and C++ code bases (Section 3.2); we confirm this with the developers as well (Section 5).

We present our research hypotheses in Section 2, followed by data collection and statistical methodology in Section 3. We first investigated whether code bases are shifting from C to C++ and found that this shift occurs for all but one application (Section 4.1). We then compared internal qualities for code bases in each language and could confirm that C++ code has higher internal quality than C code (Section 4.2). We found the same trend for external quality, i.e., that C++ code is less prone to bugs than C code (Section 4.3). Finally, we found that C++ code takes less maintenance effort than C code (Section 4.4).

To our knowledge, this is the first study that compares programming languages while controlling for variations in both developer expertise and development process, and draws *statistically significant* conclusions. In summary, our main contributions are:

- A novel way to analyze factors that impact software quality while controlling for both developer expertise and the software development process.
- A multi-faceted software evolution study of four large applications, measuring software quality, development effort, and code base shifts between languages.
- A formulation of four hypotheses and statistical analyses designed to capture whether a particular language leads to better software.

2. RESEARCH HYPOTHESES

Our study is centered around four research hypotheses designed to determine whether C++ (a higher-level programming language) produces better software than C (a lower-level language):

H1: C++ is replacing C as a main development language. At the beginning of the development process for an application, the best-suited language is chosen as the primary language. Later on, developers might decide to replace the primary language, e.g., if the potential benefits of migrating to a new language outweigh the costs. Our hypothesis is that, as the advantages of C++ become apparent, applications that have started with C as their primary language are shifting to C++. To verify this, we measured the change in percentage of C and C++ code over an application's lifetime; if the C++ percentage increases over time, we can conclude that C is being replaced by C++.

H2: C++ code is of higher internal quality than C code. One of the trademarks of high-level languages is that they enable the construction of software that displays higher internal quality than software written in low-level language, i.e., software that is less complex, easier to understand and easier to change. To test this hypothesis, for each application, we computed normalized code complexities for C and C++ using several metrics. If the hypothesis held, we should observe that, on average, C++ code is less complex than C code.

H3: C++ code is less prone to bugs than C code. Software bugs are due to a variety of reasons, e.g., misunderstood requirements, programmer error, poor design. The programming language plays a key role in preventing bugs; for example, polymorphic functions can avoid code cloning and copy-paste errors, and strongly-typed language eliminate many potential runtime errors. We use this reasoning to postulate our next hypothesis: due to the higher-level features, C++ code is less bug-prone than C code.

H4 : C++ code requires less effort to maintain than C code. Computing the effort that goes into software development and maintenance is difficult, especially for open-source projects, where the development process is less structured than in commercial settings [28]. Our findings indicate that even when there is no explicit allocation of tasks to developers, most developers contribute to both the C and C++ code bases. Our hypothesis is that the effort required to maintain and extend the C++ code base is lower than the effort associated with the C code base.

3. METHODOLOGY AND DATA SOURCES

We ran our empirical study on four popular open source applications written in a combination of C and C++. We used several criteria for selecting our test applications. First, since we are interested in long-term software evolution and pursue statistically significant results, the applications had to have a long release history. Second, applications had to be sizable, so we can understand the issues that appear in the evolution of realistic, production-quality software. Third, the applications had to be actively maintained by a large number of developers. Fourth, the applications had to be used by a wide number of users who report bugs and submit patches.

Table 1 presents high-level data on application evolution. The second and third columns show the time span we considered for each application and the number of official releases. The rest of the columns contain information (version, date and size) for the first and last releases. The period in column 2 is different from the time interval between the first and last release dates of the application (as reported in columns 6 and 9), because we could find bug and file change information that predate the first official releases.

3.1 Applications

We now provide a brief overview of each application.

Firefox (`http://www.mozilla.com/firefox`) is the second most widely-used web browser [14]. Originally named *Phoenix*, it was renamed to *Firebird*, and then renamed to Firefox in 2004. We considered Phoenix and Firebird in our study because the application's source code remained unchanged after the renamings. Firefox is mostly written in C and C++; it also contains HTML and JavaScript code that contribute to less than 3% of the total code.

Blender (`http://www.blender.org`) is a 3D content creation suite, available for all major operating systems. It is mostly written in C and C++; it also has a Python component that contributes to less than 2% of the total code. We used the source code available in the svn repository for our analyses [6].

VLC (`http://www.videolan.org`) is a popular [42], cross-platform open-source multimedia framework, player and server maintained by the VideoLAN project.

MySQL (`http://www.mysql.com`) is a popular [31] open

Application	Period (years)	Releases	First official release			Last official release		
			Version	Date	Size (LOC)	Version	Date	Size (LOC)
Firefox	1998-2010	92	0.1 (Phoenix)	02/09/2000	1,976,860	3.6	1/21/2010	3,780,122
Blender	2001-2009	28	2.27	09/25/2003	253,972	2.49b	09/03/2009	1,144,641
VLC Media Player	1998-2009	83	0.1.9	03/06/2000	144,159	1.0.2	09/19/2009	293,736
MySQL	2000-2009	13	3.23	10/01/2001	815,627	6.0	12/05/2009	991,326

Table 1: Application information.

source relational DBMS. MySQL was first released internally in 1995, followed by a publicly available Windows version in 1998. In 2001, with version 3.23, the source code was made available to the public. Therefore, for measuring internal quality and maintenance effort, we consider 13 major and minor releases since 3.23. Our external quality measurements depend on the bug databases of the applications; for MySQL, the database stores bug and patch reports for major releases 3.23, 4.1, 5.0, 5.1, and 6.0 only, thus our external quality findings for MySQL are confined to major releases only.

3.2 Data Collection

We now describe our data collection methodology. We first checked out the source code of all official releases from the version control management systems the applications use, then collected file change histories, and finally extracted bug information from the application-specific bug databases.

Committer distribution. An explicit goal of our study was to look at C and C++ code that was part of the *same* project, to keep most factors of the software development process constant. One such factor is developer expertise; anecdotal evidence suggests that expertise greatly affects software quality [7]. Ideally, to understand the difference between the C and C++ languages, we need to study code written by developers who are proficient in both C and C++. In Figure 1 we plot the percentages of developers who contribute to C++ code base only (top area), C code base only (bottom area) and to both C and C++ code bases (middle area). We observe that a large percentage of developers contribute to both C and C++ code. To verify that developers in the middle area commit in equal measures to both code bases, we selected random versions from each application. We then compared the mean values for the C commits and C++ commits for all those developers who commit to both code bases. We found that the mean values for C and C++ commits are comparable (using *Welch's t*-test as explained in Section 3.3), i.e., most developers commit in equal measures to both code bases. This ensures that we effectively control for developer competence, and any changes to software attributes (e.g., quality) can be attributed to the underlying programming language only. In Section 5 we present further evidence against selection bias, i.e., that perceived task difficulty and developer competence do not determine language choice.

Dividing source code into C and C++ groups. Identifying whether a file belongs to the C code base or the C++ code base is not trivial, because header files often use the extension ".h" for both C and C++ headers, while ".hpp" or ".hh" extensions are reserved for C++ headers. We considered a header file as a C++ header file *if and only if* all the files it is included in are C++ files; otherwise we consider it as a C header file. The implementation files were divided based on extension: ".c" for C files, and ".cpp" or ".cc" for C++ files.

Collecting file change histories. For testing hypotheses 3 and 4 we need precise information about bugs and code changes

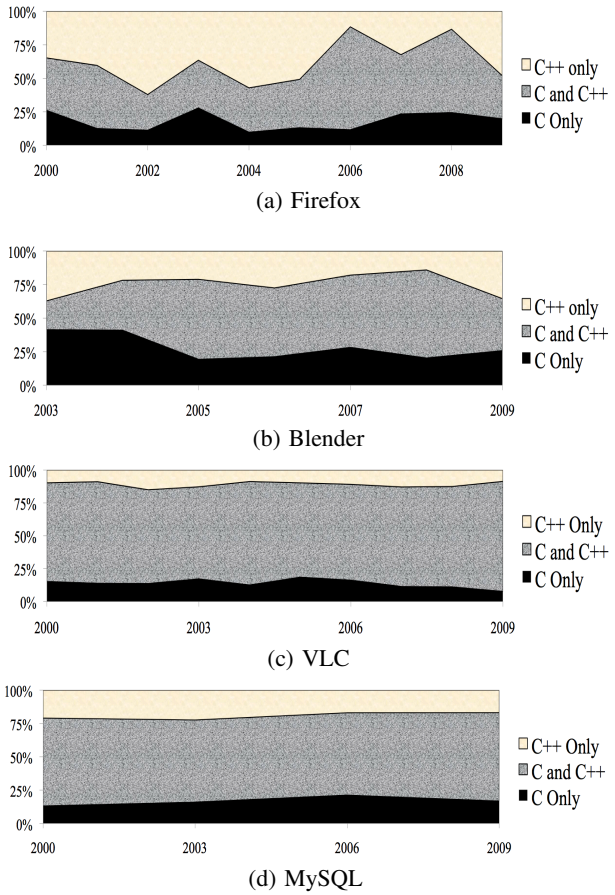


Figure 1: Committer Distribution.

associated with each version. We obtain this information by analyzing change logs associated with source files, after dividing files into C and C++ groups. Note that it is not sufficient to extract change histories for files in the last version only, because some files get deleted as the software evolves; rather, we need to perform this process for each version.

Accurate bug counting. We use defect density to assess external quality. Collecting this information is non-trivial, due to incomplete information in bug databases. As we explain shortly, to ensure accuracy, we cross-check information from bug databases[1] with bug information extracted from change logs. One problem arises from bugs assigned to no particular version; for instance, 33% of the fixed bugs in Firefox are not assigned to a specific Fire-

[1]Defect tracking systems vary: Firefox uses the Bugzilla database [8], Blender uses it own tracker [4], VLC uses Trac [41], and MySQL uses Bazaar [2] and Launchpad [24].

fox version in the Bugzilla database. This problem is compounded in applications which exhibit parallel evolution, as the co-existence of two or more parallel development branches makes version assignment problematic. Another problem is that, often, for bug fixes that span several files, the bug databases report only a partial list of changed files. However, if we search for the bug ID in the change logs, we get the complete list of files that were changed due to a particular bug fix. Therefore, we used both bug databases and change logs as bug data sources. We used a two-step approach for bug counting. First, we searched for keywords such as "bug", "bugs", "bug fixes", and "fixed bug", or references to bug IDs in log files; similar methods have been used by other researchers for their studies [39, 15, 29]. Second, we cross-checked our findings from the log files with the information in the databases to improve accuracy, similar to techniques used by other researchers for computing defect density or fault prediction [20, 39]). With the bug information at hand, we then associate a certain bug to a certain version: we used release tags, dates the bug was reported, and commit messages to find the version in which the bug was reported in, and we attributed the bug to the previous release.

Extracting effort information. To measure maintenance effort, we counted the number of commits and the churned eLOC [2] (sum of the added and changed lines of code) for each file for a release, in a manner similar to previous work by other researchers [32, 13]. This information is available from the log files.

3.3 Statistical Analysis

Variations in release frequency. Our applications have different release frequencies: Firefox, VLC, and Blender have pre-releases (alpha or beta) before a major release, while MySQL has major releases only. Differences in release frequency and number of official versions (more than 80 for Firefox and VLC, 27 for Blender and 13 for MySQL) lead to an imbalance while performing statistical analyses across all applications and could affect our study. In particular, if we allowed the values for Firefox and VLC to dominate the sample size, then the net results would be biased towards the mean of the values in the Firefox and VLC sample sets. To preserve generality and statistical significance, we equalize the sample set sizes as follows: for each official release date, we construct an observation for each application; the value for each observation is either actual, or linearly interpolated from the closest official releases, based on the time distance between the actual release and the day of the observation. This procedure ensures that we have an equal number of observations for all applications and eliminates bias due to varying release frequencies. To ensure that the interpolated values do not introduce noise in our sample, we tested whether the original sample sets are normally distributed by using the *Kolmogorov–Smirnov* normality test. [3] We found that most of our original data sets are normally distributed and hence we can safely add the interpolated values to our sample.

Hypothesis testing. We perform statistical hypothesis testing to validate our analyses and the conclusions we draw. We use the *t*-test method to analyze our samples. For instance, if we have two sample sets, A and B, the *t*-test predicts the probability that a randomly chosen value from set A will be greater, lesser or equal to a randomly chosen value in set B. Although our sample sizes are equal, their variances differ, and therefore we use a special case of *t*-test called *Welch's t*-test [43]. For the rest of the paper, by *t*-test we mean *Welch's t*-test. The *t*-test returns a *t*-value for a fixed level of statistical significance and the mean values of each of the sample sets. In our study we only consider 1% statistically significant *t*-values, to minimize chances of *Type I* error.[4] According to standard *t*-test tables, the results are statistically significant at the 1% level if $t\text{-value} \geq 2.08$. In our case, we compute the values for a particular metric for both C and C++ code bases and perform a *t*-test on the individual sample sets. For example, if for a certain metric, the *t*-test returns a $t\text{-value} \geq 2.08$ and the mean of the C sample set is *greater* than the mean of the C++ sample set, we claim a *statistical significance of 1%*; that is, if a value is chosen *randomly* from the C sample set, there is a 99% probability that the value of the random variable chosen will be closer to the mean of the C sample set than to the mean value of the C++ sample set. For each hypothesis testing, we report the mean of each sample set from C and C++ code bases, the *t*-values and the degrees of freedom, *df*.[5] We perform regression analysis for testing hypothesis *H1*, where we report the *p*-value, which is analogous to the *t*-value for the *t*-tests. For 1% statistically significant results, we must have $p\text{-value} \leq 0.01$.

Application	First release		Last release	
	C (%)	C++ (%)	C (%)	C++ (%)
Firefox	25.08	74.91	20.13	79.86
Blender	47.84	52.15	77.52	22.47
VLC	98.65	1.35	79.86	20.14
MySQL	49.82	50.17	40.35	59.64

Table 2: Percentage of C and C++ code.

4. STUDY

In this section we discuss each hypothesis, the metrics we use to test it, as well as our findings. For conciseness, we only present selected graphs for each hypothesis; however, interested readers can refer to our technical report [3] for the complete set of graphs.

4.1 Code Distribution

Hypothesis (H_A^1): C++ is replacing C as a main development language.

Metrics. To test this hypothesis we study how the percentages of C and C++ code change as an application evolves. We measure the eLOC of C and C++ code for each version using the Resource Standard Metrics (RSM) tool [36]. If the hypothesis holds, we should find that C++ percentages increase over an application's lifetime.

Results. In Figure 2 and Table 2 we show the changes in C and C++ code percentages. To verify whether, over time, the code base is shifting from C to C++, we perform a statistical hypothesis testing. Our null hypothesis, H_0^1, is that, over time, the code base division between C and C++ either remains constant, or the percentage of C code increases. We perform a two-step statistical analysis to verify this: (1) we measure the difference δ in the percentages of both C and C++ code ($\delta = \%\text{C++} - \%\text{C}$), and (2) we perform a linear regression analysis, where the independent variable is time (number of days since first release) and the dependent variable is

[2]Effective lines of code (eLOC) are those lines that are not comments, blanks or standalone braces or parentheses.

[3]The *Kolmogorov–Smirnov* test is used for testing the normality of a distribution.

[4]A *Type I* error occurs when an acceptable hypothesis is rejected.

[5]Degrees of freedom is the number of independent observations in a sample of data that are available to estimate a parameter of the population from which that sample is drawn.

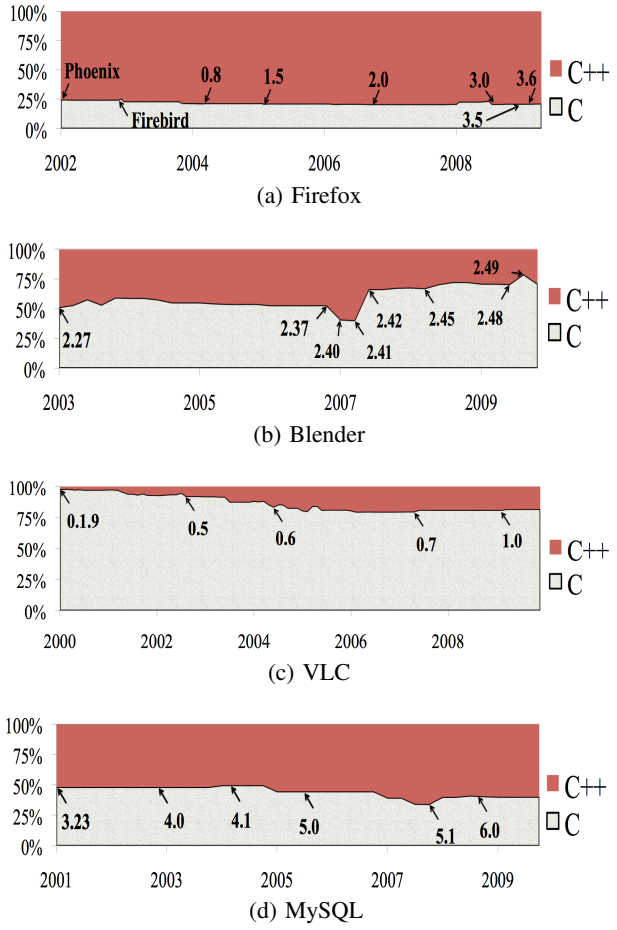


Figure 2: eLOC distribution per language.

δ. If H_0^1 is true, we should find that $\beta \leq 0$; if H_0^1 is rejected, we should have $\beta > 0$. We first perform this hypothesis testing across all applications (as described in Section 3.3) and then for each individual application. We present the results of the hypothesis testing in Table 3 when measured across all applications. Since we have $\beta > 0$ and p-value ≤ 0.01, we reject the null hypothesis H_0^1. Therefore, when performing the analysis across all applications, we observe that the primary code base is shifting from C to C++, i.e., H_A^1 is confirmed. In Table 4, we present the results for applications when tested in isolation. We observe that we can reject H_0^1 for all applications except Blender. The p-values presented for H_0^1 do not imply that, for a given version of an application, the percentage of C++ code in that version is higher than the percentage of C code; rather, they imply that if a version of an application is chosen at random, there is a 99% probability that the percentage of C++ code in that version will be higher than in previously released versions of the same application.

Conclusion. Using linear regression, we confirmed that the percentage of C code is decreasing over time. However, when considering Blender in isolation, we notice a decrease in the percentage of C++ code, which is also evident from Figure 2(b) and Table 2. The reason behind the increase in the percentage of C code in Blender, as explained by one of the main developers [5], is that the developers "try to keep sections in the same code they were originally developed in."

Criterion	Conclusion
H_0^1	H_0^1 is **rejected** at 1% significance level (β = 0.0036, p-value = 0.0002, df = 702)
H_A^1	H_A^1 is **accepted** (% of C code is decreasing over time, while % of C++ code is increasing)

Table 3: Hypothesis testing for shift in code distribution (*H1*).

Application	β	p-value (1% significance)	df	Conclusion for H_0^1
Firefox	0.0019	0.00049	97	Rejected
Blender	−0.0196	0.00001	27	Not rejected
VLC	0.0118	0.0007	72	Rejected
MySQL	0.0041	0.00262	11	Rejected

Table 4: Application-specific hypothesis testing for shift in code distribution (*H1*).

4.2 Internal Quality

Hypothesis (H_A^2): C++ code is of higher internal quality than C code.

Metrics. Internal quality is a measure of how easy it is to understand and maintain an application. For each file in each application version, we use RSM to compute two standard metrics: cyclomatic complexity [6] and interface complexity. [7] As pointed out by Mockus et al. [28], normalizing the absolute value of a metric by dividing it by total eLOC is problematic. Since only a fraction of the code changes as the application evolves, normalized values become artificially lower as the size of the source code increases. In our case, we found that the distributions of complexity values (across all files, for a particular application version) are skewed, thus arithmetic mean is not the right indicator of an ongoing trend. Therefore, to measure complexity for a specific version, we use the *geometric mean* computed across the complexity values for each file in that version. These geometric mean values constitute the sample sets for our hypothesis testing.

Results. Our null hypothesis is that C code has lower or equal code complexity compared to C++. To test this, we formulate two null sub-hypotheses corresponding to each complexity metric:

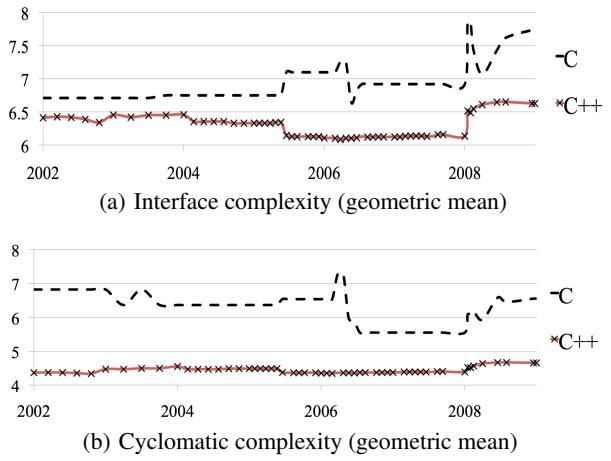
H_0^{c1}**:** The cyclomatic complexity of C code is less than or equal to the cyclomatic complexity of C++ code.

H_0^{c2}**:** The interface complexity of C code is less than or equal to the interface complexity of C++ code.

If we reject hypotheses H_0^{c1} and H_0^{c2}, we conclude that the cyclomatic and interface complexities of C code are greater than those of C++ code. We perform two t-tests on each hypothesis: across all applications, and on individual applications. The results of both t-tests are presented in Tables 5, 6, and 7. Since the t-values are greater than 2.08, both when measured across all applications and when considering the projects in isolation, we could reject both null sub-hypotheses. Moreover, as can be seen in Tables 5, 6, and 7, the mean values for the C sets are significantly higher than the mean values for the C++ sets.

[6] Cyclomatic complexity is the number of logical pathways through a function [26].

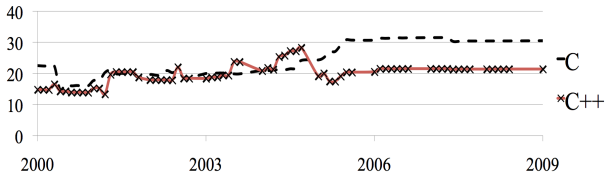
[7] Interface complexity is the sum of number of input parameters to a function and the number of return states from that function [37].



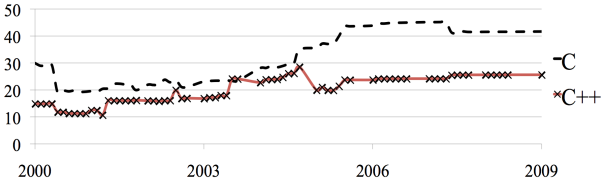
(a) Interface complexity (geometric mean)

(b) Cyclomatic complexity (geometric mean)

Figure 3: Internal Quality in Firefox.



(a) Interface complexity (geometric mean)



(b) Cyclomatic complexity (geometric mean)

Figure 4: Internal Quality in VLC.

We now discuss application-specific changes we noticed during this analysis. For VLC (Figure 4)[8] we find that, although initially both C and C++ have similar complexity values, starting in 2005, the interface complexity of C++ code decreases, while the interface complexity for C increases. However, for Firefox (Figure 3), Blender, and MySQL, the complexity of C code is always greater than that of C++ code.

Conclusion. For all the applications we consider, we could confirm that C++ code has *higher* internal quality than C code.

4.3 External Quality

Hypothesis (H_A^3): C++ code is less prone to bugs than C code.

Metrics. External quality refers to users' perception and acceptance of the software. Since perception and acceptance are difficult to quantify, we rely on *defect density* as a proxy for external quality. Similar to Mockus et al. [28], we use two metrics for defect density: number of defects divided by the total eLOC and number of defects divided by the change in eLOC (ΔeLOC). As discussed by Mockus et al., number of defects per total eLOC is potentially problematic as only a fraction of the original code changes in the new version of an application. Measuring defects over ΔeLOC is thus a good indicator of how many bugs were introduced in the newly added code.

[8]To increase legibility, we deleted the markers for some minor or maintenance releases from Figures 3–8. The actual values are reflected in graph curves, hence this *does not affect* our results and analyses.

Criterion	Conclusion
H_0^{c1}	H_0^{c1} is **rejected** at 1% significance level ($\|t\| = 5.055$ when df = 354) Mean values: C = 16.57 , C++ = 11.02
H_0^{c2}	H_0^{c2} is **rejected** at 1% significance level ($\|t\| = 3.836$ when df = 387) Mean values: C = 16.52 , C++ = 12.63
H_A^2	H_A^2 is **accepted** (C++ code is of higher internal quality than C.)

Table 5: *t*-test results for code complexities (*H2*) across all applications.

Application	Mean values		$\|t\|$ (1% significance)	df
	C	C++		
Firefox	6.44	4.46	3.19	146
VLC	31.64	20.41	8.73	144
Blender	7.19	4.11	5.29	54
MySQL	28.17	21.52	2.11	12

Table 6: Application-specific *t*-test results for cyclomatic complexity.

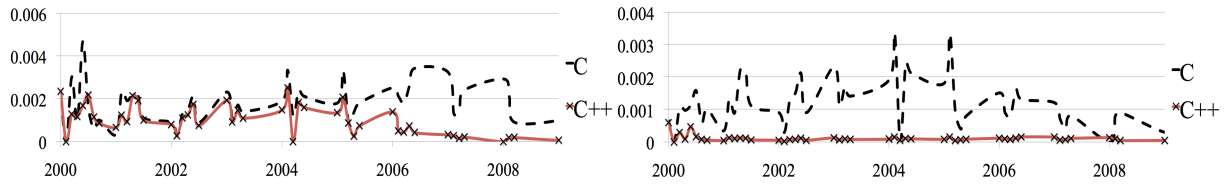
Results. Our null hypothesis is: "C code has lower or equal defect density than C++ code." Based on the metrics we use to measure defect density, we divide the main hypothesis into two null sub-hypotheses:

H_0^{d1}**:** The defect density (measured over ΔeLOC) for C code is less than or equal to defect density in C++ code.

H_0^{d2}**:** The defect density (measured over total eLOC) for C code is less than or equal to defect density in C++ code.

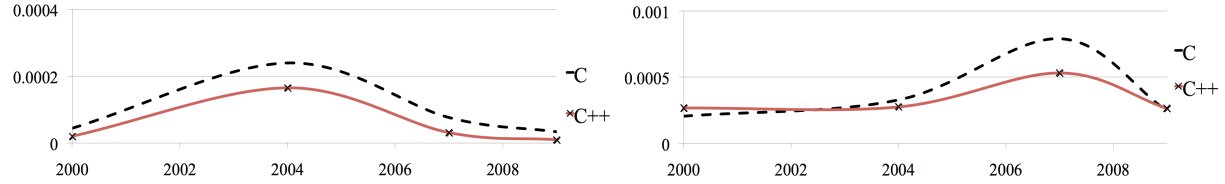
Similar to *t*-tests for code complexity, we perform two sets of *t*-tests: one across all applications, and another, for each application individually, using the original values. We present the results of the two tests in Tables 8, 9, and 10. From the *t*-values and differences in the mean defect densities of C and C++, we could reject both null sub-hypotheses when measured across all applications. When we apply the *t*-test using absolute values of defect densities for individual applications, we reject the null hypothesis at a statistically significant level for all programs except MySQL. This is caused by the unavailability of bug information for minor MySQL releases (which results in a small sample size for MySQL only) and does not affect the statistical significance of our conclusions, i.e., accepting H_A^3. As can be seen in Tables 8, 9, and 10, the mean defect density values for the C sets can be up to *an order of magnitude* higher than the mean values for the C++ sets.

In Figure 5 we present the evolution of defect densities in VLC (Firefox is similar [3]), and note that these values tend to oscillate. The oscillations are due to bugs in major releases; these bugs tend to be subsequently fixed in maintenance releases. In MySQL we found that, for the first version only, the defect density of C++ code is slightly higher than the defect density of C code (when measured over total eLOC, see Figure 6(b)); this is not the case for subsequent versions. In Blender, we found that C code had higher defect



(a) Defect Density over ΔeLOC (b) Defect Density over total eLOC

Figure 5: Defect Density in VLC.



(a) Defect Density over ΔeLOC (b) Defect Density over total eLOC

Figure 6: Defect Density in MySQL.

Application	Mean values		$\|t\|$ (1% significance)	df
	C	C++		
Firefox	6.78	6.18	9.45	188
VLC	28.20	22.92	3.61	144
Blender	13.80	5.86	16.63	54
MySQL	27.71	17.13	3.41	22

Table 7: Application-specific *t*-test results for interface complexity.

Criterion	Conclusion
H_0^{d1}	H_0^{d1} is **rejected** at 1% significance level ($\|t\| = 4.77$ when df = 482) Mean values: C = 0.109 , C++ = 0.015
H_0^{d2}	H_0^{d2} is **rejected** at 1% significance level ($\|t\| = 4.82$ when df = 489) Mean values: C = 0.04 , C++ = 0.006
H_A^3	H_A^3 is **accepted** (C++ code is less prone to bugs than C code.)

Table 8: *t*-test results for defect density (*H3*) across all applications.

density than C++, for both metrics; we omit the graphs for brevity.

Conclusion. Based on the *t*-test results, we could confirm H_A^3, that is, C++ code is less prone to bugs than C code.

4.4 Maintenance Effort

Hypothesis (H_A^4): C++ code requires less effort to maintain than C code.

Metrics. Prior work [13, 22, 21] has indicated that measuring software maintenance effort, or building effort estimation models for open source software is non-trivial, due to several reasons, e.g., the absence of organizational structure, developers working at their leisure. A widely used metric for effort is the number of commits divided by total eLOC [16, 21]. To avoid considering those parts of code which remain unchanged in a new release (similar to the argument presented for measuring defect density in Section 4.3), we also measure number of commits divided by ΔeLOC.

Application	Mean values		$\|t\|$ (1% significance)	df
	C	C++		
Firefox	0.02607	0.01939	1.0417	139
VLC	0.00178	0.00093	5.0455	87
Blender	0.03246	0.01981	1.7077	54
MySQL	0.00012	0.00007	0.6080	4

Table 9: Application-specific *t*-test results for defect density over ΔeLOC.

Application	Mean values		$\|t\|$ (1% significance)	df
	C	C++		
Firefox	0.43246	0.16294	2.6412	106
VLC	0.00119	0.0001	9.8210	49
Blender	0.00551	0.00128	4.5520	30
MySQL	0.00046	0.00036	0.5190	3

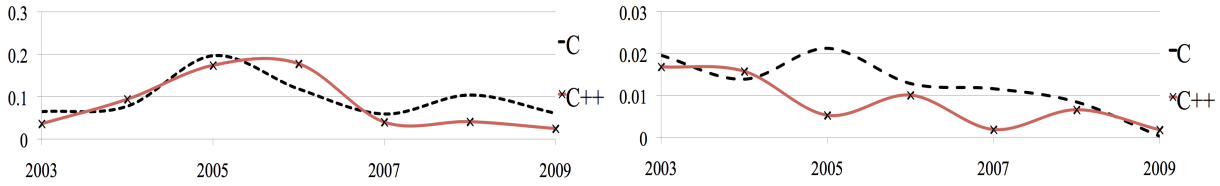
Table 10: Application-specific *t*-test results for defect density over total eLOC.

Results. Our null hypothesis is: "C files require less or equal effort to maintain than C++ files." We divide this into two null sub-hypotheses using the effort metrics we discussed:

H_0^{e1}: The maintenance effort (measured over ΔeLOC) for C files is less than, or equal to, the maintenance effort for C++ files.

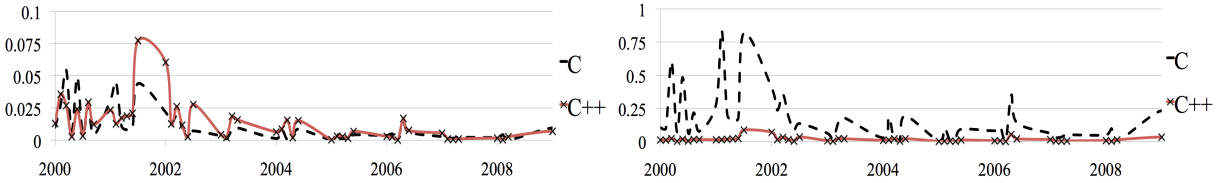
H_0^{e2}: The maintenance effort (measured over total eLOC) for C files is less than, or equal to, the maintenance effort for C++ files.

When we perform the *t*-test across all applications, we could *not* reject our null hypothesis H_0^{e1} at 1% significance level, as shown in Table 11. This is due to the very small difference between the mean values of effort for C and C++ files when measured over ΔeLOC. However, we could reject our null hypothesis H_0^{e2} and confirm that the effort to maintain C files (when measured over total eLOC) is higher than the effort for C++ files. Note that we could reject H_0^{e1}



(a) Effort over ΔeLOC (b) Effort over total eLOC

Figure 7: Maintenance Effort for Blender.



(a) Effort over ΔeLOC (b) Effort over total eLOC

Figure 8: Maintenance Effort for VLC.

Criterion	Conclusion
H_0^{e1}	H_0^{e1} is **not rejected** at 1% significance level ($\|t\| = 1.218$ when df = 147) Mean values: C = 1.07 , C++ = 0.999 H_0^{e1} is **rejected** at 10% significance level
H_0^{e2}	H_0^{e2} is **rejected** at 1% significance level ($\|t\| = 2.455$ when df = 102) Mean values: C = 0.594 , C++ = 0.26

Table 11: *t*-test results for maintenance effort (*H4*) across all applications.

Application	Mean values		$\|t\|$ (1% significance)	df
	C	C++		
Firefox	4.1780	2.9626	1.0824	15
VLC	0.1719	0.0154	5.4499	49
Blender	0.1026	0.0919	1.7077	54
MySQL	0.0004	0.0002	0.6080	4

Table 12: Application-specific *t*-test results for maintenance effort over ΔeLOC.

at a weaker level of significance (10%), but, to retain uniformity and reduce the probability of introducing errors in our conclusions, we employ 1% level of significance across all hypothesis testing.

In Tables 12 and 13 we present our *t*-test results on H_0^{e1} and H_0^{e2} for individual applications. While we could only reject the null sub-hypotheses for VLC, note that the mean values for C are *higher* than the mean values for C++ for all applications.

From Figures 7 and 8 we notice how the file maintenance effort changes over time for VLC and Blender. As evident from the mean values from Tables 12 and 13, even though for one version the absolute value for effort for C++ files might be higher than C, across the whole evolution period, the maintenance effort values for C files is higher than the effort required to maintain C++ files.

Conclusion. We could confirm our hypothesis only when measuring maintenance effort over total eLOC. When measuring maintenance effort over ΔeLOC, even though the mean values for C++ files are less than the mean values for C files, we could not validate our hypothesis at a statistically significant level.

Application	Mean values		$\|t\|$ (1% significance)	df
	C	C++		
Firefox	2.4110	0.9217	4.9284	12
VLC	0.0119	0.0104	0.5180	95
Blender	0.0114	0.0069	1.2568	9
MySQL	0.0017	0.0012	1.0737	3

Table 13: Application-specific *t*-test results for effort maintenance over total eLOC.

5. THREATS TO VALIDITY

Selection Bias. An important trait of our study is aiming to reduce selection bias, i.e., making sure that high-level languages do not appear to be "better" because they are favored by more competent developers, or are used for easier tasks. Therefore, following our quantitative analysis, we also asked developers several questions to determine whether there is bias in language selection. For example, a key VLC developer stated [11] that "developers are expected to know C and C++" when they join the project and perceived difficulty of implementing a task "does not really [play a role in selecting the language]." Moreover, for the VLC project, LUA, a high-level language, is preferable to C: "LUA is used most for text processing, where performance is not critical, and C would be too prone to programming errors [...] Where performance is not an issue [...] C code has and will continue to be replaced with LUA code." Perceived task difficulty does not play a role in language selection in Blender either, as indicated in Section 4.1.

Empirical Studies. The empirical nature of our study exposes it to construct, content, internal and external threats to validity.

Construct validity relies on the assumption that our metrics actually capture the intended characteristic, e.g., defect density accurately models external quality, source code metrics accurately model internal quality. We intentionally used multiple metrics for each hypothesis to reduce this threat. We randomly chose several versions from each application and verified that, for those devel-

opers who commit to both code bases, the number of C commits and C++ commits are comparable. This balance indicates developers have no selection bias towards which code base they want to commit to—an assumption confirmed by developers.

To ensure *content validity* we selected applications that contain both C and C++ code, written by developers who contribute to both code bases, and we analyzed as long a time span in a program's lifetime as possible. For Firefox, we do not count bugs labeled as "invalid bugs," though we found 7 instances (out of 5786 Firefox bugs) where these bugs were re-opened in subsequent versions. There is a possibility that invalid bugs might be re-opened in the future, which will very slightly change our results.

Internal validity relies on our ability to attribute any change in system characteristics (e.g., metric values or eLOC) to changes in the source code, rather than accidentally including or excluding files, inadvertently omitting bugs or commits. We tried to mitigate this threat by (1) manually inspecting the releases showing large gains (or drops) in the value of a metric, to make sure the change is legitimate, and (2) cross-checking the change logs with information from bug databases as described in Section 3.2. When we group committers by the code base they are contributing to, we use committer IDs to assign developers to the C code base, to the C++ code base, or to both code bases. Since we cannot differentiate among committers who have multiple IDs, we run the risk of over-reporting or under-reporting the number of committers.

External validity, i.e., the results generalize to other systems, is also threatened in our study. We have looked at four open-source projects written in a combination of C and C++ to keep factors such as developer competence or software process uniform. Therefore we cannot claim that arbitrary programs written in C++ are of higher quality than arbitrary programs written in C; nevertheless, we show that all other factors being equal, the choice of programming language does affect quality. It is also difficult to conclude that our proposed hypotheses hold for proprietary software, or for software written in other combinations of lower- and higher-level languages, e.g., C and Java or C and Ruby.

6. RELATED WORK

Myrtveit et al. [30] performed an empirical study to test if using C++ (as the primary programming language) increased developer productivity when compared to using C. They used projects written either in C or C++ (no description of the projects were provided), computed effort as the number of hours a developer worked on a project and found that language choice has no effect on productivity. Phipps [35] conducted a study using two different small projects (one in Java and the other in C++, developed by the author himself) to compare the effects of programming language on defect density and developer productivity. This study found that defect density was unaffected by the programming language and using Java the author was twice as productive as when using C++ (even though he is more experienced in C++ than Java). Paulk [34], Jones et al. [19], and Lipow et al. [25] studied factors that affect software quality; they infer that there is *no correlation* between software quality and the programming language used in building software; we now discuss how our study differs, and why our conclusions are different from theirs. Jones et al. [19] used a functionality-based size measure (the eLOC required to implement a function point) and concluded that the only factor that affects software quality is the number of function points in a program. We choose interface complexity as one of the metrics for internal code quality, e.g., if file A has more function calls with more parameters than file B, A's interface complexity is higher than B's. Thus, similar to Jones et al., our metric effectively relates functions to code complexity and software quality. Lipow et al. [25] found that program size affects software quality, but code quality is unaffected by the choice of the programming language. The authors studied applications written in different languages but implementing the same functionality; they do not control for programmer expertise. Jones et al. and Lipow et al. do not provide a measure of goodness of fit for their analyses. Paulk [34] compared small applications written in different languages by graduate and upper undergraduate students and found that, in these applications, software quality was more dependent on programmer abilities than on the programming language. Their conclusion strengthens our case, i.e., the need to control for programmer competence. In contrast to all these studies, our study examines real-world, large applications, written and maintained by seasoned developers competent in both languages.

Fateman [12] discusses the advantages of Lisp over C and how C itself contributes to the "pervasiveness and subtlety of programming flaws." The author categorizes flaws into various kinds (logical, interface and maintainability) and discusses how the very design of C, e.g., the presence of pointers and weak typing, makes C programs more prone to flaws. Using Lisp obviates such errors, though there exists a (much smaller) class of bugs specific to Lisp programs. The author concludes that Lisp is still preferable to C. We consider C and C++ to study the difference between a lower-level, and (comparatively) higher-level, language. Our goal was not to identify those C features that are more error prone and how C++ helps avoid such errors. Rather, our analysis is at a higher level, i.e., we analyze which language (C or C++) helps produce code that is less complex, less buggy and requires less effort to maintain.

Holtz et al. [17] compare four languages (C, Pascal, Fortran 77, and Modula-2) to identify how language syntax and semantics affect software quality. They compare how easy it is to understand a program written in different languages, and how this facilitates development and maintenance. For example, various control constructs (e.g., recursion, `while` loops, etc.) offered by different programming languages can increase or decrease code size, understandability and code complexity. In contrast, we study applications as a whole, rather than with respect to language constructs. Burgess et al. [9] and Wichmann et al. [44] examine how the choice of programming language may affect software quality, by focusing on programming language constructs, similar to Holtz et al. However, the authors do not perform any empirical study to differentiate between languages, and do not provide any statistical results.

Hongyu et al. [18] compared code complexity with software quality to test the influence of the language used, but their study is limited to applications written in a single language (C, C++, or Java) by different teams of students and conclude that software quality depends on developer expertise only. In contrast, our study looks at complexity and quality in mixed C/C++ applications where the same developers contribute to both C and C++ code bases, hence developer expertise is kept constant while varying the language.

7. CONCLUSIONS

In this paper we introduce a novel methodology for quantifying the impact of programming language on software quality and developer productivity. To keep factors such as developer competence or software process uniform, we investigate open source applications written in a combination of C and C++. We formulate four hypotheses that investigate whether using C++ leads to better software than using C. We test these hypotheses on large data sets to ensure statistically significant results. Our analyses demonstrate that applications that start with C as the primary language are shifting their code base to C++, and that C++ code is less complex, less prone to errors and requires less effort to maintain.

In future work, we plan to investigate how specific language constructs lead to differences in software quality. We also plan to broaden our analysis by comparing applications written in combinations of other languages, e.g., C and Ruby. Finally, it would be interesting to test our hypotheses on commercial software.

Acknowledgments

We thank M Squared Technologies LLC for providing Resource Standard Metrics for our project. We thank Sudipto Das, Dennis Jeffrey, Changhui Lin, Markus Lumpe, Christian Bird, Arnab Mitra, Rajdeep Sau, the anonymous ICSE mentor, and the anonymous referees for their helpful comments on this paper. We also thank Remi Denis-Courmont from the VLC project, Kent B. Mein Jr. from Blender and David Miller from Mozilla for help with data collection and answering useful questions related to our study.

8. REFERENCES

[1] F. B. e. Abreu and W. Melo. Evaluating the impact of object-oriented design on software quality. In *METRICS '96*.

[2] Bazaar. http://bazaar.launchpad.net/~mysql/mysql-server/.

[3] P. Bhattacharya and I. Neamtiu. Higher-level Languages are Indeed Better: A Study on C and C++. Technical report, University of California, Riverside, March 2010. http://www.cs.ucr.edu/~pamelab/tr.pdf.

[4] Blender Bug Tracker. https://projects.blender.org/tracker.

[5] Blender Forum. http://www.blender.org/forum/viewtopic.php?t=16694.

[6] https://svn.blender.org/svnroot/bf-blender/tags/.

[7] F. P. Brooks, Jr. *The Mythical Man-Month*. Addison-Wesley, 1975.

[8] Bugzilla. https://bugzilla.mozilla.org/.

[9] C. J. Burgess. Software quality issues when choosing a programming language. Technical report, University of Bristol, UK, 1995.

[10] DedaSys LLC. Programming Language Popularity. http://www.langpop.com/.

[11] R. Denis-Courmont. Personal communication, based on experience with VLC development, June 2010.

[12] R. Fateman. Software Fault Prevention by Language Choice: Why C is Not My Favorite Language. Technical report, University of California, Berkeley, 2000.

[13] J. Fernández-Ramil, D. Izquierdo-Cortazar, and T. Mens. What does it take to develop a million lines of open source code? In *OSS*, pages 170–184, 2009.

[14] Firefox Statistics. http://www.computerworld.com/s/article/9140819/1_in_4_now_use_Firefox_to_surf_the_Web.

[15] M. Fischer, M. Pinzger, and H. Gall. Populating a release history database from version control and bug tracking systems. In *ICSM '03*.

[16] A. Hars and S. Ou. Working for free? motivations for participating in open-source projects. *Int. J. Electron. Commerce*, 6(3):25–39, 2002.

[17] N. M. Holtz and W. J. Rasdorf. An evaluation of programming languages and language features for engineering software development. *Engineering with Computers*, 1988.

[18] Z. Hongyu, Z. Xiuzhen, and G. Ming. Predicting defective software components from code complexity measures. In *ISPRDC '07*.

[19] C. Jones. Backfiring: Converting lines-of-code to function points. *Computer*, pages 87–88, 1995.

[20] S. Kim, T. Zimmermann, K. Pan, and E. J. J. Whitehead. Automatic identification of bug-introducing changes. In *ASE*, pages 81–90, 2006.

[21] S. Koch. Exploring the effects of coordination and communication tools on the efficiency of open source projects using data envelopment analysis. *IFIP*, 2007.

[22] S. Koch. Exploring the effects of sourceforge.net coordination and communication tools on the efficiency of open source projects using data envelopment analysis. *Empirical Softw. Eng.*, 14(4):397–417, 2009.

[23] J. Koskinen. Software maintenance costs, Sept 2003. http://users.jyu.fi/~koskinen/smcosts.htm.

[24] Launchpad. https://launchpad.net/mysql-server.

[25] M. Lipow. Number of faults per line of code. *TSE'82*.

[26] T. J. McCabe. A complexity measure. In *ICSE'76*.

[27] T. Mens, M. Wermelinger, S. Ducasse, S. Demeyer, R. Hirschfeld, and M. Jazayeri. Challenges in software evolution. In *IWPSE '05*, pages 13–22, 2005.

[28] A. Mockus, R. T. Fielding, and J. D. Herbsleb. Two case studies of open source software development: Apache and mozilla. *ACM Trans. Softw. Eng. Methodol.*, 11(3), 2002.

[29] A. Mockus and L. G. Votta. Identifying reasons for software changes using historic databases. ICSM'00, pages 120–130.

[30] I. Myrtveit and E. Stensrud. An empirical study of software development productivity in C and C++. In *NIK'08*.

[31] MySQL Statistics. http://www.mysql.com/why-mysql/marketshare/.

[32] N. Nagappan and T. Ball. Use of relative code churn measures to predict system defect density. In *ICSE*, 2005.

[33] NIST. The economic impacts of inadequate infrastructure for software testing. Planning Report, May 2002.

[34] M. C. Paulk. *An Empirical Study of Process Discipline and Software Quality*. PhD thesis, Univ. of Pittsburgh, 2005.

[35] G. Phipps. Comparing observed bug and productivity rates for java and c++. *Software Practice and Experience*, 29, April 1999.

[36] M Squared Technologies - Resource Standard Metrics. http://msquaredtechnologies.com/.

[37] RSM Metrics. http://msquaredtechnologies.com/m2rsm/docs/index.htm.

[38] R. C. Seacord, D. Plakosh, and G. A. Lewis. *Modernizing Legacy Systems: Software Technologies, Engineering Process and Business Practices*. Addison-Wesley, 2003.

[39] J. Śliwerski, T. Zimmermann, and A. Zeller. When do changes induce fixes? In *MSR '05*, pages 1–5, 2005.

[40] TIOBE Software BV. TIOBE Programming Community Index. http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html.

[41] Trac. http://trac.videolan.org/.

[42] VLC Statistics. http://www.videolan.org/stats/downloads.html.

[43] B. L. Welch. The generalization of "student's" problem when several different population variances are involved. *Biometrika*, 1947.

[44] B. A. Wichmann. Contribution of standard programming languages to software quality. In *Software Engineering Journal*, pages 3–12, 1994.