Journal of Software: Evolution and Process

JOURNAL OF SOFTWARE: EVOLUTION AND PROCESS
J. Softw. Evol. and Proc. 2016; **28**:589–618
Published online 9 December 2015 in Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/smr.1760

Empirical analysis of the relationship between CC and SLOC in a large corpus of Java methods and C functions

Davy Landman[1,*,†], Alexander Serebrenik[2], Eric Bouwers[3] and Jurgen J. Vinju[1,2,4]

[1]*Centrum Wiskunde & Informatica, Amsterdam, The Netherlands*
[2]*Eindhoven University of Technology, Eindhoven, The Netherlands*
[3]*Software Improvement Group, Amsterdam, The Netherlands*
[4]*INRIA Lille Nord Europe, Lille, France*

ABSTRACT

Measuring the internal quality of source code is one of the traditional goals of making software development into an engineering discipline. Cyclomatic complexity (CC) is an often used source code quality metric, next to source lines of code (SLOC). However, the use of the CC metric is challenged by the repeated claim that CC is redundant with respect to SLOC because of strong linear correlation.

We conducted an extensive literature study of the CC/SLOC correlation results. Next, we tested correlation on large Java (17.6 M methods) and C (6.3 M functions) corpora. Our results show that linear correlation between SLOC and CC is only moderate as a result of increasingly high variance. We further observe that aggregating CC and SLOC as well as performing a power transform improves the correlation.

Our conclusion is that the observed linear correlation between CC and SLOC of Java methods or C functions is not strong enough to conclude that CC is redundant with SLOC. This conclusion contradicts earlier claims from literature but concurs with the widely accepted practice of measuring of CC next to SLOC. Copyright © 2015 John Wiley & Sons, Ltd.

Received 19 February 2015; Revised 10 August 2015; Accepted 21 October 2015

KEY WORDS: McCabe cyclomatic complexity; empirical validation; software maintenance; metrics

1. INTRODUCTION

In previous work [1], one of the authors analyzed the potential problems of using the cyclomatic complexity (CC) metric to indicate or even measure source code complexity per Java method. Still, because understanding code is known to be a major factor in providing effective and efficient software maintenance [2], measuring the complexity aspect of internal source code quality remains an elusive goal of the software engineering community. In practice, the CC metric is used on a daily basis for this purpose precisely, next to another metric, namely source lines of code (SLOC) [3, 4].

There exists a large body of literature on the relation between the CC metric and SLOC. The general conclusion from experimental studies [5–8] is that there exists a strong linear correlation between these two metrics for arbitrary software systems. The results are often interpreted as an incentive to discard the CC metric for any purpose that SLOC could be used for as well, or as an incentive to normalize the CC metric for SLOC.

At the same time, the CC metric appears in every available commercial and open-source source code metrics tool, for example, http://www.sonarqube.org/, and is used in the daily practice of software

*Correspondence to: Davy Landman, Centrum Wiskunde & Informatica, Amsterdam, The Netherlands.
†E-mail: davy.landman@cwi.nl

assessment [4] and fault/effort prediction [9]. This avid use of the metric directly contradicts the evidence of strong linear correlation. Why go through the trouble of measuring CC?

Based on the related work on the correlation between CC and SLOC, we have the following working hypothesis:

Hypothesis 1
There is strong linear (Pearson) correlation between the CC and SLOC metrics for Java methods and C functions.

Note that the current paper both includes and extends the contributions of a previously published conference paper, which focused on the Java language [10]. The new contributions of this paper are as follows:

- construction of a big C corpus,
- corpus-based analysis of CC/SLOC correlation for another programming language (C),
- a significantly extended literature study,
- comparison between the results for C and Java,
- more detailed study of possible explanations for higher correlation after aggregation on the file level, and
- study of the possibly confounding effect of the size of the corpora.

We studied a C language corpus because it is most representative of the languages analyzed in literature, and we could construct a large corpus based on open-source code. Java is an interesting case next to C as it represents a popular modern object-oriented language, for which we could also construct a large corpus. A modern language with a comparable but significantly more complex programming paradigm than C, such as Java, is expected to provide a different perspective on the correlation between SLOC and CC.

Both for Java and C, our results of investigating the strong correlation between CC and SLOC are negative, challenging the external validity of the experimental results in literature as well as their interpretation. The results of analyzing a linear correlation are not the same for our (much larger) corpora of modern Java code that we derived from Sourcerer [11] and C code derived from the packages of Gentoo Linux. Similarly, we observe that higher correlations can only be observed after aggregation to the file level or when we arbitrarily remove the larger elements from the corpus. Based on analyzing these new results, we will conclude that CC cannot be discarded based on experimental evidence of a linear correlation. We therefore support the continued use of CC in industry next to SLOC to gain insight in the internal quality of software systems for both the C and the Java languages.

The interpretation of experimental results of the past is hampered by confusing differences in definitions of the concepts and metrics. In the following, Section 2, we therefore focus on definitions and discuss the interpretation in related work of the evidence of correlation between SLOC and CC. We also identify six more hypotheses. In Section 3, we explain our experimental setup. After this, in Section 4, we report our results, and in Section 5, we interpret them before concluding in Section 6.

2. BACKGROUND THEORY

In this section, we carefully describe how we interpret the CC and SLOC metrics, we identify related work, and introduce the hypotheses based on differences observed in related work.

2.1. Defining SLOC and CC

Although defining the actual metrics for lines of code and CC used in this paper can be easily performed, it is hard to define the concepts that they actually measure. This lack of precisely defined dimensions is an often lamented, classical problem in software metrics [12, 13]. The current paper does not solve this problem, but we do need to discuss it in order to position our contributions in the context of related work.

First, we define the two metrics used in this paper.

Definition 1
SLOC A line of code is any line of program text that is not a comment or blank line, regardless of the number of statements or fragments of statements on the line. This specifically includes all lines containing program headers, declarations, and executable and non-executable statements [14, p. 35].

Definition 2
CC The cyclomatic complexity of a program[‡] is the maximum number of linearly independent circuits in the control flow graph of said program, where each exit point is connected with an additional edge to the entry point [15].

As explained by McCabe [15], the CC number can be computed by counting forks in a control flow graph and adding 1, or equivalently counting the number of language constructs used in the abstract syntax tree (AST), which generate forks ('if', 'while', etc.), and adding 1.

This last method is the easiest and therefore preferred method of computing CC. Unfortunately, which AST nodes generate decision points in control flow for a specific programming language is not so clear because this depends on the intrinsic details of programming language semantics. The unclarity leads to metric tools generating different values for the CC metric, because they count different kinds of AST nodes [16]. Also, derived definitions of the metric exist, such as 'extended cyclomatic complexity' [17] to account for a different way of computing CC. Still, the original definition by McCabe is sufficiently general. If we interpret it based on a control flow graph, it is applicable to any programming language, which has subroutines to encapsulate a list of imperative control flow statements. Section 3 describes how we compute CC for C and Java.

Note that we include the Boolean && and || operators as conditional forks because they have short-circuit semantics in both Java and C, rendering the execution of their right-hand sides conditional. Still, this is not the case for all related work. For completeness sake, we therefore put the following hypothesis up for testing as well:

Hypothesis 2
The strength of linear correlation between CC and SLOC of neither Java methods nor C functions is significantly influenced by including or excluding the Boolean operators && and ||.

We expect that exclusion of && and || does not meaningfully affect correlations between CC and SLOC, because we expect Boolean operators not to be used often enough and not in enough quantities within a single subroutine to make a difference.

2.2. Literature on the correlation between CC and SLOC

We have searched methodically for related work that experimentally investigates a correlation between CC and SLOC. This results, to the best of our knowledge, in the most complete overview of published correlation figures between CC and SLOC to date. Our previous literature study [10] resulted in 15 relevant papers obtained by scanning the titles of 600 papers. For the current overview, we scanned the full text of 326 new papers identifying 18 new relevant papers.

In our previous literature study, we used Google Scholar to find all papers citing Shepperd's paper from 1988 [12], which also investigate Hypothesis 1. Furthermore, in the same study, we scanned the titles of the 200 most relevant search results[§] for papers citing McCabe's original paper [15] and matching the 'Empirical' search query.

The previous literature study can be seen as a restricted form of snowballing [18]. To extend our coverage of the literature, and correct for limitations of snowballing [19], we combine snowballing and systematic literature review (SLR). We formulated the PICO criteria inspired by the SLR guidelines of Kitchenham and Charters [20]:

[‡]In this context, a 'program' means a subroutine of code like a procedure in Pascal, function in C, method in Java, subroutine in Fortran, and program in COBOL. From here on, we use the term 'subroutine' to denote either a Java method or a C function.
[§]Google Scholar's sort by relevancy.

Population Software
Intervention CC or Cyclomatic or McCabe
Comparison SLOC or LOC or Lines of Code
Outcomes Correlation or Regression or Linear or R^2

Ideally, following the Kitchenham and Charters' guidelines [20], we should have constructed a query using the PICO criteria: 'Software and (CC or Cyclomatic or McCabe) and (SLOC or LOC or Lines of Code) and (Correlation or Regression or Linear or R^2)'. Unfortunately, Google Scholar does not support nested conditional expressions. Therefore, we have used the PICO criteria to create $1 \times 3 \times 3 \times 4 = 36$ different queries producing 24 K results. Because Google Scholar sorts the results on relevancy, we chose to read only the first two pages of every query, leaving 720 results.

After noise filtering and duplication removal, 326 papers remained, containing 11 of the 15 papers included in our previous literature study [10].

Together, we systematically scanned the full text of these papers, using the following inclusion criteria:

1. Is the publication peer-reviewed?
2. Is SLOC or LOC measured?
3. Is CC measured (possibly as weight in Weighted Methods per Class [21])?
4. Is Pearson correlation or any other statistical relation between SLOC and CC reported?
5. Are the measurements performed on method, function, class, module, or file level (higher levels are ignored)?

Using this process, we identified 18 new papers. The resulting 33 papers are summarized in Table I.

The SLR guidelines require the inclusion and the search queries to be based on the title, abstract, and keywords. We deviated from this because for the current study, we are interested in a reported relation between SLOC and CC, whether the paper focuses on this relation or not. This required us to scan the full text of each paper, which the Kitchenham and Charter process does not cater for. Note that Google Scholar does index the body of papers.

The result of the previous process is summarized by the multi-page Table I. All levels and corpus descriptions in the table are as reported in the original papers: the interpretation of these might have subtle differences, for example, Module and Program in Fortran could mean the same. Because the original data are no longer available, it is not possible to clarify these differences. The variables mentioned in the correlation column are normalized as follows. If all lines in a unit (file, module, function, or method) were counted, LOC was reported. If comments and blank lines were ignored, SLOC was reported. If the line count was normalized on statements, we reported logical lines of code. We normalized R to R^2 by squaring it whenever R was originally reported.

Figure 1 visualizes the R^2 from the related work in Table I grouped by language and aggregation level. Most related work reports R^2 higher than 0.5, and there is not a clear upward or downward trend over the years. The only observable trends are that newer work (after 2000) predominantly performed aggregation on a file level (with the notable exception of four papers [8, 22, 42, 44]) and that while the early studies have been mostly conducted on Fortran, the most common languages analyzed after 2000 are Java and C.

In the rest of this section, we will formulate hypotheses based on observations in the related work: different aggregation methods (Section 2.3), data transformations (Section 2.4), and the influence of outliers and other biases in the used corpora (Section 2.5).

2.3. Aggregating CC over larger units of code

Cyclomatic complexity applies to control flow graphs. As such, CC is defined when applied to code units, which have a control flow graph. This has not stopped researchers and tool vendors to sum the metric over larger units, such as classes, programs, files, and even whole systems. We think that the underlying assumption is that indicated 'effort of understanding' per subroutine would add up to indicate total effort. However, we do not clearly understand what such sums mean when interpreted back as an attribute of control flow graphs, because the compositions of control flow graphs that these sums should reflect do not actually exist.

Table I. Overview of related work on CC and SLOC up to 2014.

Year	Level	Correlation	Language	Corpus	R^2	Comments
◦1979 [23]	Subroutine	SLOC vs CC	Fortran	27 programs with SLOC ranging from 25 to 225	*0.65 0.81	The first result is for a CC correlation on subroutine level, and the second result is on a program level.
◦1979 [5]	Program	SLOC vs CC	Fortran	27 programs with SLOC ranging from 36 to 57	0.41	
◦1979 [6]	Program	log(LLOC) vs log(CC)	PL/1	197 programs with a median of 54 statements	*0.90	
◦1979 [24]	Subroutine	LOC vs CC	Fortran	26 subroutines	*0.90	
◦1980 [25]	Module	LOC vs CC	Fortran	10 modules, 339 SLOC	0.90	
◦1981 [26]	Module	SLOC vs CC	Fortran	25.5 K SLOC over 137 modules	0.65	
◦1984 [7]	Module	SLOC vs CC	Fortran	517 code segments of one system	0.94	No correlation between module SLOC and module CC. Then the authors grouped modules into five buckets (by size) and calculated the average CC per bucket. Over these *five data points*, they reported the high correlation.
◦1987 [27]	Program	SLOC vs CC	Fortran	255 student assignments, range of 10 to 120 SLOC	0.82	Study comparing 31 metrics, showing histogram of the corpus, and scatter plots of selected correlation.
1987 [28]	Module	SLOC vs CC	S3	Two subsystems with 67 modules	0.83 0.87	After a power transform on the first subsystem, the R^2 increased to 0.89.
◦1989 [29]	Routine	SLOC vs CC	Pascal and Fortran	1 system, 4.5 K routines, 232 K SLOC Pascal, 112 K SLOC Fortran	*0.72 0.70	The first result was for Pascal, the second for Fortran.
1989 [30]	Procedure	SLOC vs CC	Pascal	1 stand-alone commercial system, 7 K procedures	*0.96	
1990 [31]	Program	LOC vs CC	COBOL	311 student programs	*0.80	
1990 [32]	Module	LOC vs CC	Pascal	981 modules from 27 course projects	0.40	10% outliers were removed.
◦1991 [33]	Module	SLOC vs CC	Pascal and COBOL	19 systems, 824 modules, 150 K SLOC	0.90	The paper also compared different variants of CC.
◦1993 [34]	Program	LOC vs CC	COBOL	3 K programs	*0.76	
1997 [35]	File	SLOC vs CC	COBOL	600 modules of a commercial system	*0.79	
2000 [9]	Module	LOC^2 vs CC	Unreported	380 modules of an Ericson system	0.62	Squaring the LOC variable was performed as an argument for the nonlinear relationship.

(Continues)

Table I. (Continued)

Year	Level	Correlation	Language	Corpus	R^2	Comments
2000 [36]	File	LOC vs CC	C and DLSs	1.5 M LOC subsystem of telephony switch, 2.5 K files	0.94	
2001 [37]	Class	SLOC vs CC	C++	174 classes	0.77	A study discussing the confounding factor of size for OO metrics, WMC is a sum of CC for the methods of a class.
2001 [38]	File	LOC vs CC	RPG	293 programs 200 K LOC	0.86	
2005 [39]	Module	LOC vs CC	Pascal	41 small programs	0.59	The programs analyzed were written by the authors with the sole purpose of serving as data for the publication.
2006 [40]	File	LOC vs CC	C	NASA JM1 data set, 22 K files, 11 K LOC	0.71	
2007 [41]	File	LOC vs CC	C and C++	77 k small programs	0.78	The corpus contains multiple implementation of 59 different challenges. Outliers were removed based on the CC variable. The correlation was calculated after calculating the mean per challenge.
2007 [42]	Function	SLOC vs CC	C	XMMS project, 109 K SLOC over 260 files	*0.51	
2007 [43]	File	log(SLOC) vs log(CC)	C	FreeBSD packages, 694 K files	0.87	Using the SLOC variable, 1 K files suspected of being generated code were removed.
◦2008 [44]	Diff	LOC vs CC	Java and C and C++ and PHP and Python and Perl	13 M diffs from SourceForge	0.56	The paper contains many different correlations based on the revision diffs from 278 projects. The authors observed lower correlations for C.
2009 [45]	File	SLOC vs CC	Java	4813 proprietary Java modules	0.52	
◦2009 [46]	File	log(LOC) vs log(CC)	Java and C and C++	2200 projects from SourceForge	0.78 0.83 0.73	After discussing the distribution of both LOC and CC and their wide variance, the authors calculate a repeated median regression and recalculate R^2: 0.87, 0.93, and 0.97.

(Continues)

Table I. (Continued)

Year	Level	Correlation	Language	Corpus	R^2	Comments
◦2010 [47]	File	log(SLOC) vs log(CC)	C	ArchLinux packages, 300 K Files, of which 200 K non-header files	0.59 0.69	Initially, they observed a low correlation between CC and SLOC, and further analysis revealed header files as the cause. The second correlation is after removing these. The authors show the influence of looking at ranges of SLOC on the correlation.
◦2010 [48]	Class	SLOC vs CC	Java and C++	800 K SLOC over 12 hand-picked open-source projects	0.66	
2011 [22]	Class	SLOC vs max CC and mean CC	Java	Arc data set: 234 classes	0.12 0.08	Correlations were not statistically significant.
2014 [49]	Module	LOC vs CC	C	NASA CM1 data set	0.86	

This extends Shepperd's table [12]. The correlations with a star (*) indicate correlations on the subroutine level. The ° denotes that the relation between CC and SLOC was the main focus of the paper. The statistical significance was always high, if reported, and therefore not indicated in this table (except Malhotra [22]).
CC, cyclomatic complexity; SLOC, source lines of code; LOC, lines of code; WMC, Weighted Methods per Class.

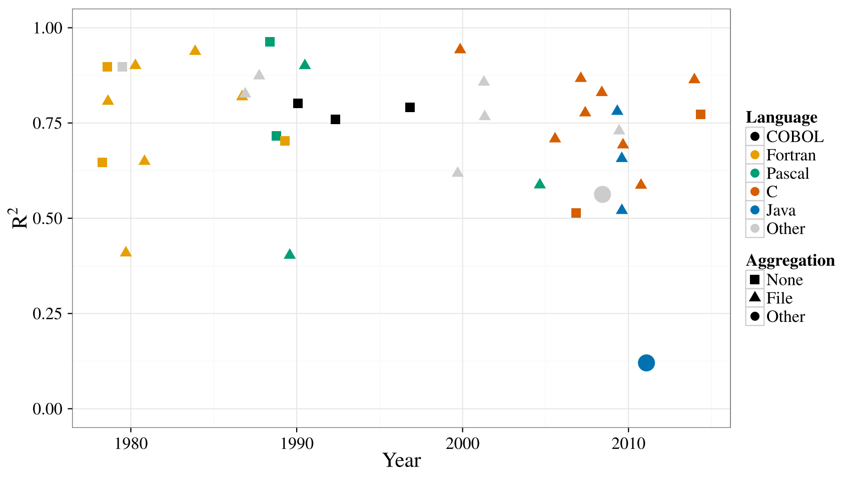


Figure 1. Visualization of the R^2 reported in related work (Table I). The colors denote the most common languages, and the shape the kind of aggregation; aggregation 'None' means that the correlation has been reported on the level of a subroutine. Note that for languages such as COBOL, the lowest level of measurement of cyclomatic complexity and source lines of code is the file level. Therefore, these are reported as an aggregation of 'None' (similar to the * indication in Table I).

Perhaps not surprisingly, in 2013, Yu *et al.* [50] found a Pearson correlation of nearly 1 between whole system SLOC and the sum of all CC. They conclude the evolution of either metric can represent the other. One should keep in mind, however, that choosing the appropriate level of aggregation is vital for validity of an empirical study: failure to do so can lead to an ecological fallacy [51] (interpreting statistical relations found in aggregated data on individual data). Similarly, the choice of an aggregation technique can greatly affect the correlation results [52–54].

Curtis and Carleton [13] and Shepherd [12] were the first to state that without a clear definition of what source code complexity is, it is to be expected that metrics of complexity are bound to measure (aspects of) code size. Any metric that counts arbitrary elements of source code sentences actually measures the code's size or a part of it. Both Curtis and Carleton, and Shepherd conclude that this should be the reason for the strong correlation between SLOC and CC. However, even though CC is a size metric, it still measures a different part of the code. SLOC measures all the source code, while CC measures only a part of the statements, which govern control flow. Even if the same dimension is measured by two metrics, that fact alone does not fully explain a strong correlation between them. We recommend the work of Abran [55], for an in-depth discussion of the semantics of CC.

Table I lists which studies use which level of aggregation. Note that the method of aggregation is *sum* in all but one of the papers reviewed. A possible explanation for strong correlations could be the higher levels of aggregation. This brings us to our third hypothesis:

Hypothesis 3

The correlation between aggregated CC for all subroutines and the total SLOC of a file is higher than the correlation between CC and SLOC of individual subroutines.

If this hypothesis is true, it would explain the high correlation coefficients found in literature when aggregated over files: it would be computing the sum over subroutines that causes it rather than the metric itself. Hypothesis 3 is non-trivial because it depends, per file, on the ratio between the size of the bodies and the amount of bodies what the influence of aggregation may be. This influence needs to be observed experimentally.

A confounding factor when trying to investigate Hypothesis 3 is the size of the code outside of the subroutines, such as import statements and class and field declarations in Java, and macro-definitions and function headers, typedefs, and structs in C. For the sake of brevity, we refer to this part of source code files as the 'header', even though this code may be spread over the file. A large variance in header size would negatively influence correlation on the file aggregation level, which may hide the effect of summing up the CC of the subroutines. We do not know exactly how the size of the header is distributed in C or Java files and how this size relates to the size of subroutines. To be able to isolate the two identified factors on correlation after aggregation, we also introduce the following hypothesis:

Hypothesis 4
The more subroutines we add up the CC for, the more this aggregated sum correlates with aggregated SLOC of these subroutines.

This hypothesis isolates the positive effect of merely summing up over the subroutines from the negative effect of having headers of various sizes. Hypothesis 4 is non-trivial for the same reasons as Hypothesis 3 is non-trivial.

2.4. *Data transformations*

Hypothesis 1 is motivated by the earlier results from the literature in Table I. Some newer results of strong correlation are only acquired after a log transform on both variables [6, 43, 46, 47]: indeed, log transform can help to normalize distributions that have a positive skew [56] (which is the case both for SLOC and for CC), and it also compensates for the 'distorting' effects of the few but enormous elements in the long tail. A strong correlation, which is acquired after log transform, does not directly warrant dismissal of one of the metrics, because any minor inaccuracy of the linear regression is amplified by the reverse log transform back to the original data. Nevertheless, the following hypothesis is here to confirm or deny results from literature:

Hypothesis 5
After a log transform on both the SLOC and CC metrics, the Pearson correlation is higher than the Pearson correlation on the untransformed data.

We note that the literature suggests that the R^2 values for transformed and untransformed data are not comparable [57, 58]. However, we do not attempt to find the best model for the relation between CC and SLOC, rather to understand the impact of log transformation as used by previous work on the reported R^2 values.

2.5. *Corpus bias*

The aforementioned log transform is motivated in literature after observing skewed long-tail distributions of SLOC and CC [43, 46, 47, 59]. On the one hand, this puts all related work on smaller data sets, which do not interpret the shape of the distributions in a different light. How to interpret these older results? Such distributions make relatively 'uninteresting' smaller subroutines dominate any further statistical observations. On the other hand, our current work is based on two large corpora (Section 3). Although this is motivated from the perspective of being as representative as possible for real-world code, the size of the corpus itself does emphasize the effects of really big elements in the long tail (the more we look, the more we find) as well as strengthens the skew of the distribution towards the smaller elements (we will find disproportionate amounts of new smallest elements). Therefore, we should investigate the effect of different parts of the corpus, ignoring either elements in the tail or ignoring data near the head:

Hypothesis 6
The strength of the linear correlation between SLOC and CC is improved by ignoring the smallest subroutines (as measured by SLOC).

Hypothesis 7
The strength of the linear correlation between SLOC and CC is improved by ignoring the largest subroutines (as measured by SLOC).

Hypothesis 6 was also inspired by Herraiz and Hassan's observation of an increasing correlation for the higher ranges of SLOC [47]. One could argue that the smallest of subroutines are relatively uninteresting, and a correlation that only holds for the more nontrivial subroutines would be satisfactory as well.

Hypothesis 7 investigates the effect of focusing on the smaller elements of the data, ignoring (parts of) the tail. Inspired by related work [32, 41, 43] that assumes that these larger subroutines can be interpreted as 'outliers'. It is important for the human interpretation of Hypothesis 1 to find out what their influence is. Although there are not that many tail elements, a linear model that ignores them could still have value.

3. EXPERIMENTAL SETUP

In this section, we discuss how the study has been set up. To perform empirical evaluation of the relation between SLOC and CC for subroutines, we needed a large corpus of such subroutines. To construct such a corpus, we have processed Sourcerer [11], a collection of 19 K open-source Java projects (Section 3.1), and Gentoo,[¶] a full Linux distribution containing 9.6 K C packages (Section 3.2). Then SLOC and CC have been computed for each method or function (subroutine) in the corpus (Sections 3.3 and 3.4). Finally, we performed statistical analysis of the data (Section 3.5).

3.1. Preparing the Java corpus

Sourcerer [11] is a large corpus of open-source Java software. It was constructed by fully downloading the source code of 19 K projects, of which 6 K turned out to be empty.

- **Remove non-Java files** While Sourcerer contains a full copy of each project's source code management (SCM), because of our focus on Java, we excluded all non-Java files.
- **Remove SCM branches** When Sourcerer was compiled, the whole SCM history was cloned. In particular, this means that multiple versions of the same system are present. However, inclusion of multiple similar versions of the same method would bias statistical analysis. Therefore, we removed all directories named /tags/, /branches/, and /nightly/, which are commonly used to indicate snapshot copies of source trees or temporarily forked development.
- **Remove duplicate projects** Sourcerer projects have been collected from multiple sources including Apache, Java.net, Google Code, and SourceForge. Based on Sourcerer's meta-data, we detected 172 projects, which were extracted from multiple sources—for example, from both SourceForge and Google Code. Similarly to removal of SCM branches, we have kept only one version of each project; in this case, we chose the largest version in bytes.
- **Manually reviewed duplicate files** We calculated the MD5 hash per file. The 278 projects containing more than 300 duplicate files (equal hash) were manually reviewed and fixed in case the duplication could be explained. Common reasons were non-standard SCM structure (different labels for tags and branches) and the code of third-party libraries. A list of the duplicate projects and manually removed directories is available online.[||]
- **Remove out-of-scope code** Finally, we have decided to remove code that is either external to the studied project or is test code. It is *a priori* not clear whether test code exhibits the same relation between SLOC and CC as non-test code. We removed all directories matching the following case-insensitive regular expression: `/[/\-]tests?\/|\/examples?\/|(third|3rd)[\-\_]?party/`. This filtering differs from the one used in the previous paper [10] and was based on insight gained during the construction of the C corpus. This is also why the numbers for Java in Section 4 differ from those reported previously [10].

Performing these steps, we have reduced the 390 corpus to 14.3 containing 13 K projects over 2 M files. The resulting corpus has been made publicly available [60].

[¶]https://www.gentoo.org/

[||]All code and data are available at http://www.cwi.nl/~landman/jsep2015/

3.2. Preparing the C corpus

We are not aware of a C corpus of size, age, and spread of domains comparable with Sourcerer. Therefore, we have constructed a new corpus based on Gentoo's Portage packages.[**]

We have chosen Gentoo because its packages cover a wide range of domains. Compared with other Linux distributions, Gentoo distributes the source code instead of pre-compiled binaries, enabling our analysis.

On October 14, 2014, the repository contained 65 K packages. The extensions of 40 K packages indicated an archive (e.g., tar.gz). The following process was used to construct our C corpus based on these packages.

- **Remove non-code packages** We filtered debug symbols, patch collections, translations, binary installers, data packages, binary packages, auxiliary files, and texlive modules.
- **Remove multiple versions** The Portage repository of Gentoo contains multiple versions of packages. We kept only the newest version of every package. Note that Portage does come with meta-data—'ebuild'—to collect the latest Gentoo packages, selecting a sub-set of the entire repository. We refrained from using this meta-data, because it is based on design decisions, which would introduce a selection bias (like hardening for security and library compatibility).
- **Extract packages** The remaining 20 K packages were unpacked, resulting in 8 M files.
- **Detect C code** C and C++ code share file extensions. Both .c and .h can contain C or C++ code. Using heuristics inspired by GitHub's linguist [61], we developed a tool to detect if a file contained either C or C++ code. The heuristics uses syntactical differences to detect C++ and differences between the often included standard library header files for C and C++. Of the 1.35 M files with C extensions, 1.02 M contained C code, and 0.33 M contained C++. We removed all the files with C++ code.
- **Remove out-of-scope code** Similarly to the preparation of our Java corpus, we have chosen to remove code that is not part of the application or library studied. We have used the exact same filter, removing the folders: tests, examples, and third party.
- **Detect duplicates** Similarly to the preparation of our Java corpus, we calculated the MD5 hash of all the files. The 223 packages containing more than 300 duplicate files were manually reviewed and fixed in case the duplication could be explained. Common reasons were failures in detecting multiple versions (90 packages), forks, and included third-party libraries.
- **Keep only related files** For the packages still containing C files, we also kept all files related to the possible compilation of the library. All other files were removed.

Performing these steps resulted in a corpus of 19 containing 9.8 K packages with 13 of C code in 809 K files. The corpus is publicly available [62].

3.3. Measuring Java's SLOC and CC

While numerous tools are available to measure SLOC and CC on a file level,[††] to perform our study, we require to calculate SLOC and CC per method and to precisely control the definition of both metrics. We use the M^3 framework [63], which is based on the Eclipse JDT,[‡‡] to parse the full Java source code and identify the methods in the corpus. This also generates full AST for each method for further analysis. Figure 2 depicts the source code of computing the CC from the AST of a method. The code recursively traverses the AST and matches the enumerated nodes, adding 1 for each node that would generate a fork in the Java control flow graph.

For SLOC, we decided not to depend on the information in the Eclipse AST (AST are not designed for precisely recording the lexical syntax of source code). Instead, we use the AST only to locate the source code of each separate method. To compute its SLOC, we defined a grammar in RASCAL [64] to tokenize Java input into newlines, whitespace, comments, and other words. The parser

[**]https://packages.gentoo.org/
[††]For example, http://cloc.sourceforge.net/, http://www.sonarqube.org/
[‡‡]http://www.eclipse.org/jdt

```
int calcCC(Statement impl) {
   int result = 1;
   visit (impl) {
       case \if(_,_) : result += 1;
       case \if(_,_,_) : result += 1;
       case \case(_) : result += 1;
       case \do(_,_) : result += 1;
       case \while(_,_) : result += 1;
       case \for(_,_,_) : result += 1;
       case \for(_,_,_,_) : result += 1;
       case foreach(_,_,_) : result += 1;
       case \catch(_,_): result += 1;
       case \conditional(_,_,_): result += 1;
       case infix(_,"&&",_) : result += 1;
       case infix(_,"||",_) : result += 1;
   }
   return result;
}
```

Figure 2. RASCAL source code to calculate the CC of a given method. The visit statement is a combination of a regular switch and the visitor pattern. The cases pattern match on elements of the AST.

produces a list of these tokens, which we filter to find the lines of code that contain anything else but whitespace or comments. We tested and compared our SLOC metric with other tools measuring full Java files to validate its correctness.

To be able to compare SLOC of only the subroutines compared with SLOC of the entire file, we store the SLOC of each Java method body separately (Hypothesis 4).

For Java, files without method bodies, such as interface definitions, were ignored. Out of the 2 M files, 306 K were ignored because they did not contain any method bodies.

3.4. *Measuring C's SLOC and CC*

To perform our analysis on the C code, we use the Software Analysis Toolkit of the Software Improvement Group[§§]. This proprietary toolkit uses a robust analysis approach, processes over a billion SLOC per year, and forms the basis of the consultancy services of Software Improvement Group. As part of these services, the measurements performed by the toolkit are continuously validated, both by the internal development team as well as externally by the development teams of clients and third-party suppliers.

The measurement process of the Software Analysis Toolkit consists roughly of four phases: preprocessing, tokenization, scope creation, and measurements. In the first phase, preprocessor directives are removed from the source-code. This step is required to solve issues such as illustrated in Figure 3 where only one unit declaration ends up in the final binary depending on whether debug is defined. When both parts are kept two unit headers, but only a single close bracket would be used as input to the next phase. To prevent problems in the scope creation phase, that is, not being able to find the correct units, only the first code blocks of conditional preprocessor directives are kept. That is, in the code in Figure 3, only the second and the sixth lines are passed on to the next phase.

This pragmatic approach is used because running the preprocessor is prone to errors and labor intensive because of projects relaying on specific tools and versions. Moreover, choosing a representative set of system constants is often not possible and adds unnecessary complexity to the assessment process. Processing all sources in the same way reduces overhead and makes the measurement step more objective. In our experience, choosing the first preprocessor block captures most of the code and provides reliable results in assessments where the results are validated with the development teams. Because this validation step is not possible in this experiment, all files, which after processing contain unbalanced curly braces, are removed from the corpus.

In the second phase, the code is tokenized using an internally developed tokenizer. The resulting list of tokens is used in the scope creation phase to extract a scope tree containing subroutines, modules, and packages (depending on the language). For C, the token list is inspected for patterns representing the headers of subroutines (e.g., the second line in the aforementioned code) and the

[§§]http://www.sig.eu

```
1: #ifdef debug
2:   void get_string(char prefix) {
3: #else
4:   void get_string() {
5: #endif
6:   }
```

Figure 3. C code example with conditional pre-processor directives.

body blocks (the brackets on line two and six). These scope blocks are then put into an internal graph structure.

To perform the actual measurements, all nodes representing subroutines are processed by a visitor, which works on the list of tokens associated with the node. Similar to the approach for Java, SLOC is measured by identifying all lines within a function, which contain anything else than comments or whitespace. To calculate the CC, all tokens representing the keywords case, if, for, and while and the operators ||, &&, and ? are counted. Note that because we match on tokens instead of AST nodes, the while token also captures any do…while statements, making this implementation equal to the one defined for Java (Figure 2).

C code is split over .c and .h files. Herraiz and Hassan [47] ignored all headers files (.h), but we did include them. The reason is that for C, although it is a less common idiom, putting functions in a header file is possible. Our C corpus contains 356 K header files. We chose to ignore all .c and .h files without any function bodies (similar to Java interfaces). This results in removing 331 K .h and 25 K .c files.

3.5. *Visualization and statistics methods*

Before discussing the results (Section 4), we will first discuss the chosen visualizations and statistical methods.

3.5.1. Distributions. Before comparing SLOC and CC, we describe the distributions in our data using histograms and descriptive statistics (median, mean, min, and max). The shape of distributions does have an impact on the correlation measures used, as explained earlier. All results (Section 4) should be interpreted with these distributions in mind.

3.5.2. Hexagonal scatter plots. Scatter plots with SLOC on the x-axis and CC on the y-axis represent the data in a raw form. Because of the long-tail distributions of both CC and SLOC, the data are concentrated in the lower-left quadrant of the plots, and many of the dots are placed on top of each other. Therefore, we also use log–log scatter plots. We use hexagonal scatter plots [65] to address overplotting and type I errors (false positives). The latter method divides the two-dimensional plane of the plot area in 50×50 hexagons. It then counts how many of the data points fall into each individual hexagon and uses a logarithmic 255-step grayscale gradient to color it. Compared with vanilla scatter plots, the hexagonal plots are much less confusing; the main problem is that a limited resolution on paper can create artifacts such as big black blobs of ink where in fact the raw data do not feature maximum density at all (i.e., overplotting causing type I errors). Nevertheless, it should be noted that the gradient and human perception have a limited resolution, and as such, hexagonal plots can still hide the full impact of the skewness of the distributions and the variance in the data.

3.5.3. Correlation. Most related work, if reported, uses Pearson product-moment correlation coefficient [66] (hereafter Pearson correlation), measuring the degree of linear relationship between two variables. The square of Pearson correlation is called the coefficient of determination (R^2). R^2 estimates the variance in the power of one variable to predict the other using a simple linear regression. Hereafter, we report the R^2 to describe a correlation.

Many researchers have observed that the distributions of SLOC (and CC) are right-skewed. While opinions differ on robustness of the Pearson correlation against normality violations [56, 67], a number of earlier studies attempt to compensate for the skewness of the distribution by applying a log transform and then compute the Pearson correlation [6, 43, 46].

The important matter of interpreting the results after a log transform back to the original data is discussed in Section 5.

Other researchers have transformed the data using more advanced methods in order to improve the chances for linear correlation. For example, using Box-Cox transformation [47] or performing the repeated median regression (RMR) method on a random sample [46]. Box-Cox is a power transform similar to the basic log transform. We have chosen to stick with the simpler method, following the rest of the related work, which we are trying to reproduce (Hypothesis 5).

The next method, RMR, may be useful to find some linear model, but it entails a lossy transformation. The median regression method reduces the effect of random measurement errors in the data by computing a running median. We do not have random errors in the CC or SLOC measurements, so a running median would hide interesting data. Therefore, RMR is outside the scope of this paper.

If no linear correlation is to be expected, or is found using Pearson's method, we use Spearman's rank-order correlation coefficient [68] (hereafter Spearman correlation or ρ). Similarly to the Pearson correlation, Spearman's correlation is a bivariate measure of correlation/association between two variables. However, opposed to the Pearson correlation, Spearman's correlation is employed with rank-order data, measuring the degree of monotone relationship between two variables. We apply this method only for completeness sake, because it does not generate a predictive model, which we could use to discard one of the metrics.

3.5.4. Regression. The square of Pearson's correlation coefficient is the same as the R^2 in simple linear regression. Hence, if we would find a strong correlation coefficient, we would be able to construct a good predictive linear model between the two variables, and one of the metrics would be obsolete. It is therefore important to experimentally validate the reported high correlation coefficients in literature (Table I). In general, for other correlation measures (such as Spearman's method), this relation between regression and correlation is not immediate. In particular, a strong Pearson correlation coefficient after a log transform does not give rise to an accurate linear regression model of the original data. We discuss this in more detail later when interpreting the results in Section 5.

4. RESULTS

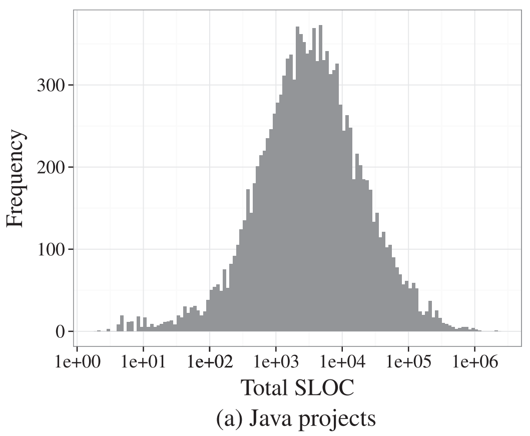
In this section, we report the results of our experiments and the statistics we applied to it. We postpone discussion of these results until Section 5.

4.1. Distributions for Java and C

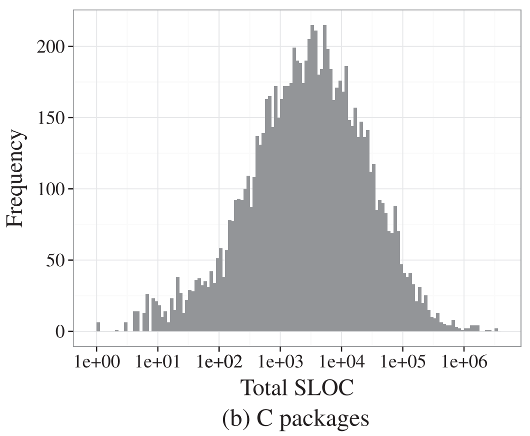
Figure 4 shows the histogram of SLOC per project, and Table II describes this distribution. The Java corpus contains 17.6 M methods spread out over 1.7 M files, and the C corpus has 6.3 M functions spread over 462 K files. The C corpus seems to have a disproportional number of packages with a low SLOC, even on the logarithmic scale. After randomly inspecting a number of packages in the range between 1 and 20 files, we concluded that next to naturally small packages, these are C files that are part of larger packages written in other languages such as Java, Python, or Perl. Lacking any argument to dismiss these files, we assume them to be just as representative of arbitrary C code as the rest.

Figure 5 shows the distribution of SLOC per Java method and C function. Table III describes their distributions. We observe skewed distributions with a long tail. To measure the degree of skewness, we calculate the moment coefficient of skewness [69], that is, the third standardized moment of the probability distribution. A positive value indicates that the right-hand tail is longer or fatter than the left-hand one. A negative value indicates the reverse. A value close to zero suggests a symmetric distribution. For our corpora, the moment coefficient of skewness equals 234.7488 for SLOC in Java and 107.277 for SLOC in C. After the log transform, it equals 1.053 for Java and 0.4042721 for C.

This means that the mean values are not at all representative for the untransformed corpora and that the smallest subroutines dominate the data. For Java, 8.8 M of the methods have 3 SLOC or fewer. This is 50% of all data points. There are 1.2 M methods with 1 or 2 SLOC; these are the methods with an empty body, in two different formatting styles or (generated) methods without newlines. The other 7.6 M methods of 3 SLOC contain the basic getters, setters, and throwers pattern frequently



(a) Java projects



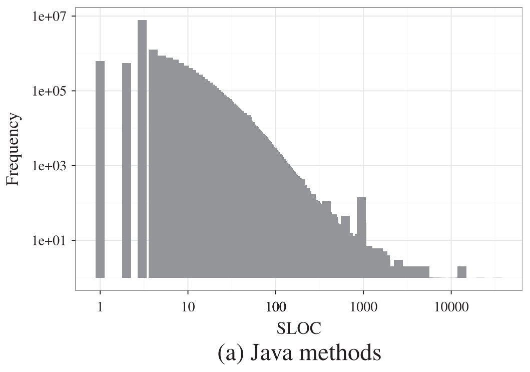
(b) C packages

Figure 4. Distribution of the non-empty projects/packages over their total source lines of code (SLOC). SLOC is on a $\log_{10}$ scale; bin width is 0.05.

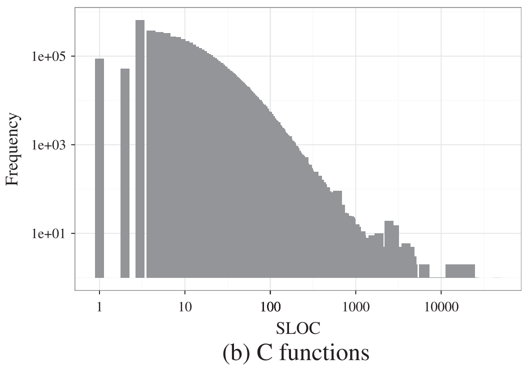
Table II. Statistics of the total SLOC per project in the corpus.

Corpus	Min.	25%	Median	Mean	75%	Max.
Java	0	1009	3219	15,270	10,250	2,207,000
C	1	671	3036	21,200	12,430	3,333,000

SLOC, source lines of code.



(a) Java methods



(b) C functions

Figure 5. Histogram of the source lines of code (SLOC) per subroutine in both corpora, in log–log space (bin width is 0.1). Here, we see that for both Java and C, small methods and functions are the most common. The bar around 1000 for Java and 3000 for C are two cases where a project contained multiple files of generated code that slightly differed per file. See Figure 7 to compare the distribution.

Table III. Descriptive statistics of the SLOC and CC per Java method and C function.

Corpus	Variable	Min.	25%	Median	Mean	75%	Max.
Java	SLOC	1	3	3	9.38	9	33,850
	CC	1	1	1	2.33	2	4377
C	SLOC	1	6	12	26.36	27	44,880
	CC	1	1	3	5.98	6	18,320

CC, cyclomatic complexity; SLOC, source lines of code.

seen in Java methods—often called one-liners. For C, this is less extreme; only 13% of the functions have an SLOC of 3 or less. The corpora differ in the strength of the skewness here: the C corpus has proportionally fewer of the smallest subroutines than the Java corpus has. Nevertheless both plots have their mode at 3 SLOC.

Figure 6 shows the distribution of CC per Java method and C function. For the Java corpus, 15.2 M methods have a CC of 3 or less. This is 86% of all data points. There are 11.6 M methods without any forks in the control flow (1 CC), that is, 65%. This observation is comparable with the 64% reported by Grechanik *et al.* for 2 K randomly chosen Java projects from SourceForge [70]. We observe that the lion's share of Java methods is below the common CC thresholds of 10 (97.00%) [15] or 15 (98.60%) [71]. The C corpus shows a comparable picture, but again with a more even distribution, which puts less emphasis on the smallest subroutines. For C, the median is at 3, while for Java, it was 1. Still, 33% of the C subroutines have a CC of 1 (straight line code). We do see that both corpora have their mode of CC at 1. For C, 85.60% functions are below the common CC threshold of 10 and 91.70% below 15.

Comparing the shape of Java's and C's distributions is complicated by the difference in corpus size. To visualize the difference in the distribution, we have used relative frequency polygons (Figure 7). These relative frequency polygons are normalized by the size of the corpus, and thus, the area under the curve is 1. This more clearly shows the difference in distribution between Java and C; for Java, there are more methods with a small SLOC and CC than C functions. The shape of the distributions is a controversial matter, which we consider outside the scope of this article.

4.2. Scatter plots

Figure 8 shows two zoomed in ($CC \leq 500$ and $SLOC \leq 1800$) hexagonal scatter plots of the subroutines in our corpus. Because of the skewed data, this figure still shows 99.98% of all data points. Figure 9 shows the same hexagonal scatter plots in a log–log space, allowing showing more

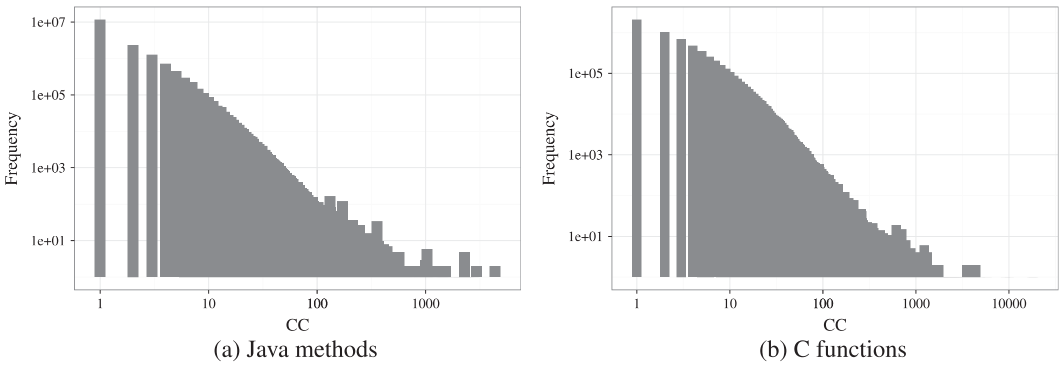


Figure 6. Histogram of the Cyclomatic Complexity (CC) per subroutine in both corpora, in log–log space (bin width is 0.1). Here, we see that for both Java and C, methods and functions with little control flow are the most common. See Figure 7 to compare the distribution.

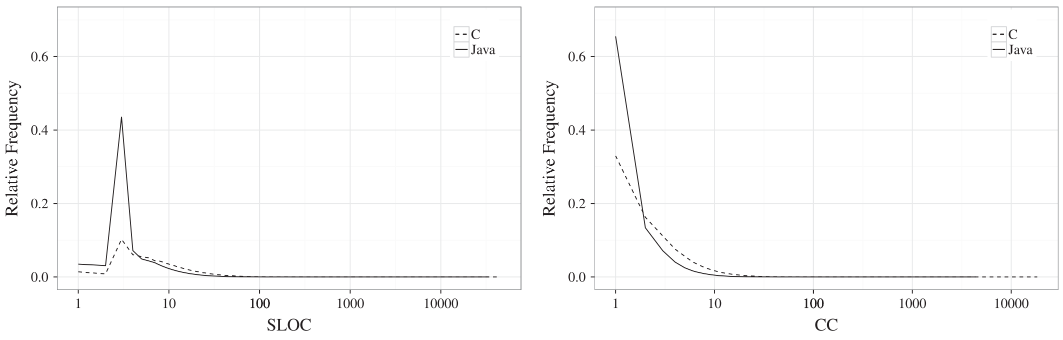
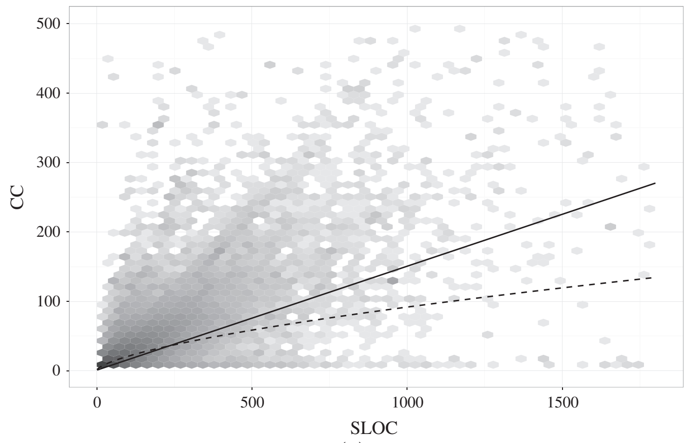
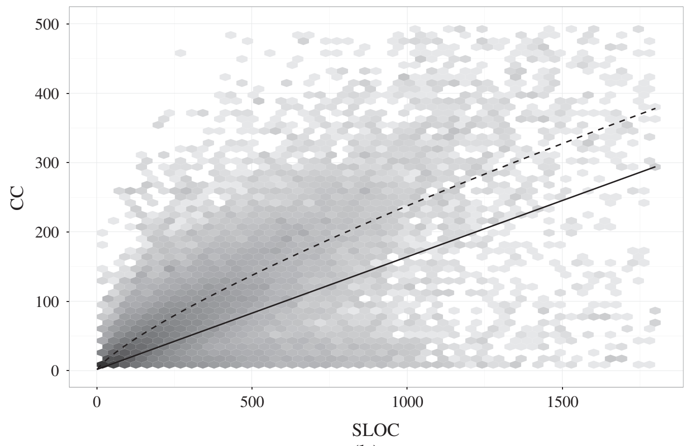


Figure 7. Relative frequency polygons for both corpora and both variables. The variables are displayed on a logarithmic scale. Relative frequency polygons are histograms normalized by the amount of data points; the area under the curve is 1. They visualize the relative difference between distributions. CC, cyclomatic complexity; SLOC, source lines of code.

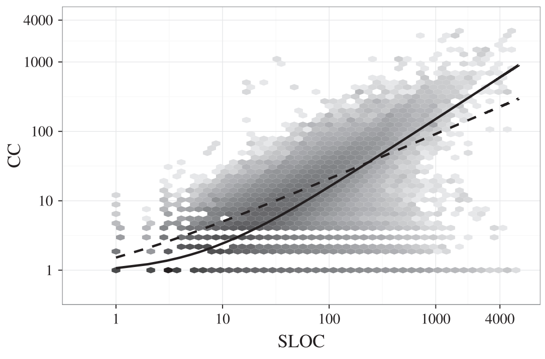


(a) Java

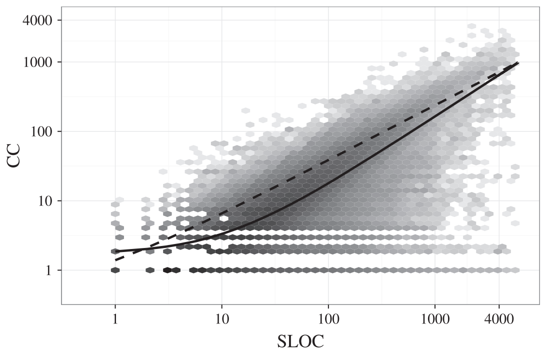


(b) C

Figure 8. Scatter plots of Source Lines of Code (SLOC) versus Cyclomatic Complexity (CC) zoomed in on the bottom left quadrant. The solid and dashed lines are the linear regression before and after the log transform. The grayscale gradient of the hexagons is logarithmic.



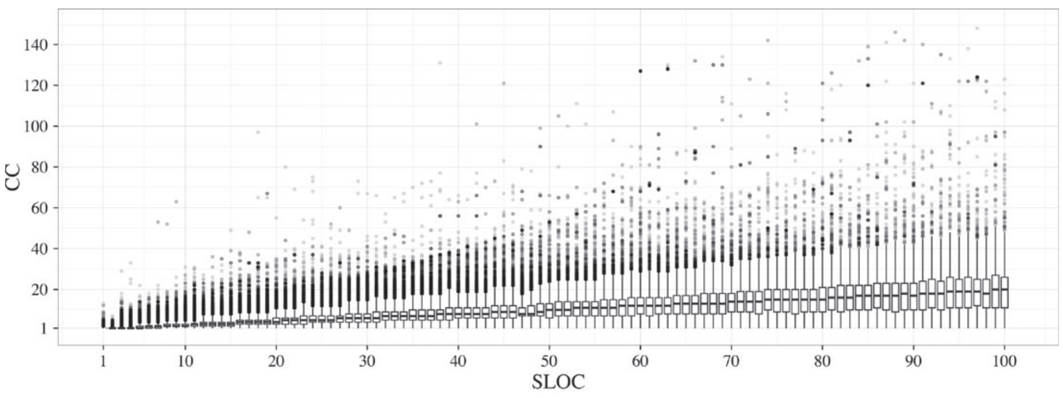
(a) Java



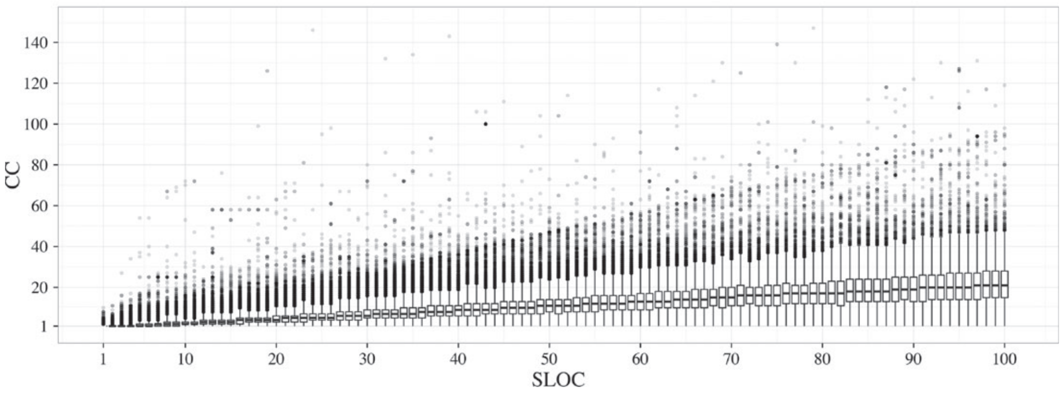
(b) C

Figure 9. Scatter plots of source lines of code (SLOC) versus cyclomatic complexity (CC) on a log–log scale. The solid and dashed lines are the linear regression before and after the log transform. The grayscale gradient of the hexagons is logarithmic.

data. The two black lines in both figures show the linear regressions before and after the log transform, which will be discussed in Section 4.3. The logarithmic grayscale gradient of the points in the scatter plot visualizes how many subroutines have that combination of CC and SLOC: the darker, the more data points. Figure 10 shows an even more zoomed in range of the scatter plots; in these box plots,



(a) Java



(b) C

Figure 10. Box plots of cyclomatic complexity (CC) per source lines of code (SLOC) on the lower range, illustrating the wide spread of Figure 9a and 9b. The median is the black line in the box; bottom and top of the box are the first and third quartile; the hinges are at the traditional 1.57 * inter-quartile range [72].

we can more clearly see the variance of CC increasing as SLOC increases. Moreover, the median is increasing, but so is the inter-quartile range. We have not created these plots for the full range of the data because these plots do not scale.

Figures 8 and 9 show a widely scattered and noisy field, with a high concentration of points in the left corner. The outline of these concentrations might hint at a positive (linear) monotone relation. However, the same outline is bounded by the minimum CC number (1) and the expected maximum CC number (CC is usually not higher than SLOC given a source code layout of one conditional statement on a single line).

We do find some points above the expected maximum CC, which we found out to be generated code and code with dozens of Boolean operators on one single line.

In previous work on the same Java corpus, we reported the same data as plotted in Figure 9a in a normal scatter plot (Figure 5 in our previous work [10]). There, we observed several ‘lines’, which might be attributed to common code idioms. The current hexagonal plot does not show these lines (see Section 3.5.2, which motivates hexagonal plots).

4.3. *Pearson correlation*

In Table IV, the first row shows the Pearson correlation over the whole corpus. The R^2 of SLOC and CC is 0.40 for Java and 0.44 for C. Figure 8a and 8b respectively depicts these linear fits, $\mathrm{CC}=0.92+0.15\cdot\mathrm{SLOC}$ and $\mathrm{CC}=1.70+0.16\cdot\mathrm{SLOC}$, as a solid black line. These R^2 are much lower than the related work in Table I, even if we focus on the related work at the subroutine/function/method level.

The Pearson correlation after a log transform showed higher numbers, which are more in line with related work that also applies a log transform [6, 43, 46, 47]. The fit for Java, the dashed line in

Table IV. Correlations for part of the tail of the independent variable SLOC.

(a) Java methods

Min. SLOC	Coverage (%)	R^2	log R^2	p	Methods
1	100	0.40	0.68	0.80	17,633,256
3	50	0.37	0.58	0.74	8,816,628
5	40	0.36	0.50	0.67	7,053,303
9	25	0.34	0.38	0.60	4,408,314
11	20	0.33	0.33	0.57	3,526,652
20	10	0.30	0.20	0.50	1,763,326
77	1	0.21	0.03	0.33	176,333
230	0.100	0.14	0.00	0.21	17,634
688	0.010	0.08	0.00	0.17	1764

(b) C functions

Min. SLOC	Coverage (%)	R^2	log R^2	p	Methods
1	100	0.44	0.71	0.83	6,259,031
12	50	0.42	0.52	0.71	3,129,516
16	40	0.41	0.47	0.68	2,503,613
27	25	0.39	0.37	0.64	1,564,758
33	20	0.38	0.33	0.62	1,251,807
56	10	0.36	0.22	0.56	625,904
218	1	0.28	0.05	0.39	62,591
703	0.100	0.20	0.01	0.30	6260
2627	0.010	0.12	0.00	0.01	626

All correlations have a high significance level ($p \leq 1 \times 10^{-16}$).
SLOC, source lines of code.

Figures 8 and 9, is $log_{10}(\text{CC}) = -0.28 + 0.65 \cdot \log_{10}(\text{SLOC}) \Leftrightarrow \text{CC} = 10^{-0.28} \cdot \text{SLOC}^{0.65}$. The fit for C (Figure 8b and 9b) is $\text{CC} = 10^{-0.41} \cdot \text{SLOC}^{0.79}$. More on the interpretation of this transform and the results is discussed in Section 5.

As discussed earlier, the data are skewed towards small subroutines and simple control flow graphs. Because 50% of Java's method and 13% of C's functions have an SLOC between 1 and 3, these points have a high influence on the correlation. We could argue that the relation—between SLOC and CC—for these smaller subroutines is less interesting. Therefore, to test Hypothesis 6, Table IV also shows the Pearson correlations for parts of the tail of the SLOC variable[¶¶]. Each row shows a different percentage of the tail of the data, and the minimum SLOC for that part.

Perhaps surprisingly, the higher the minimum SLOC (Table IV), the worse the correlation. This directly contradicts results from Herraiz and Hassan [47], who reported improving correlations for higher regions of SLOC. However, Jbara *et al.* [8] also reported decreasing correlations, except that they looked at higher CC instead of SLOC.

In three papers we cited earlier [32, 41, 43], the largest subroutines are removed from the data before calculating correlation strength, as opposed to removing the smallest subroutines (see the preceding text). To be able to compare, we report in Table V the effect of removing different percentages of the tail (related to Hypothesis 7). We mention the maximum SLOC, which is still included in each sub-set.

We further explore removing *both* the smallest and the largest subroutines. We observed that for a fixed maximum SLOC, increasing the minimum SLOC results in lower R^2 (similarly to Table IV). We further observe that for a fixed minimum SLOC, increasing the maximum SLOC results in the increase of R^2 followed by the decrease (similarly to Table V). Finally, we observe that the optimal R^2 values are obtained when no small subroutines are eliminated and the maximum SLOC is 130 for Java ($R^2 = 0.60$) and 430 for C ($R^2 = 0.67$). While the optimal R^2 values seem to be quite close, the maximum SLOC for C exceeds the maximum SLOC for Java by more than three times. This factor

[¶¶]Normal quantiles do not make sense for this data because the first few buckets would hold most of the data points for only a few of the CC and SLOC values (e.g., 1–4).

Table V. Correlations for part of the tail of the independent variable SLOC *removed*.

(a) Java methods

Max. SLOC	Coverage (%)	R^2	log R^2	ρ	Methods
33,851	100	0.40	0.68	0.80	17,633,256
934	99.995	0.53	0.68	0.80	17,632,374
688	99.990	0.54	0.68	0.80	17,631,492
230	99.900	0.59	0.68	0.80	17,615,622
77	99	0.59	0.67	0.79	17,456,923
20	90	0.51	0.55	0.74	15,869,930
11	80	0.43	0.41	0.66	14,106,604
9	75	0.37	0.32	0.60	13,224,942
5	60	0.07	0.04	0.28	10,579,953
3	50	0.00	0.00	0.02	8,816,628

(b) C functions

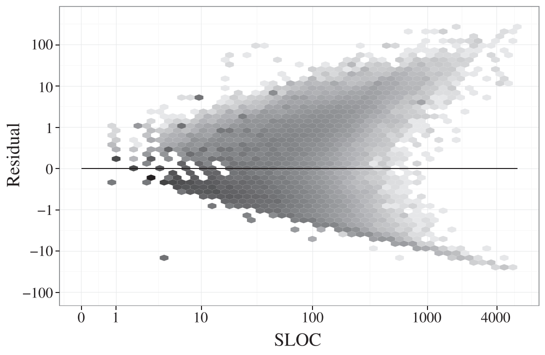
Max. SLOC	Coverage (%)	R^2	log R^2	ρ	Methods
44,881	100	0.44	0.71	0.83	6,259,031
3715	99.995	0.63	0.71	0.83	6,258,718
2622	99.990	0.63	0.71	0.83	6,258,405
703	99.900	0.67	0.70	0.83	6,252,771
218	99	0.66	0.69	0.83	6,196,440
56	90	0.56	0.61	0.80	5,633,127
33	80	0.47	0.54	0.75	5,007,224
27	75	0.44	0.50	0.73	4,694,273
16	60	0.33	0.38	0.65	3,755,418
12	50	0.26	0.29	0.59	3,129,515

All correlations have a high significance level ($p \leq 1 \times 10^{-16}$).
SLOC, source lines of code.

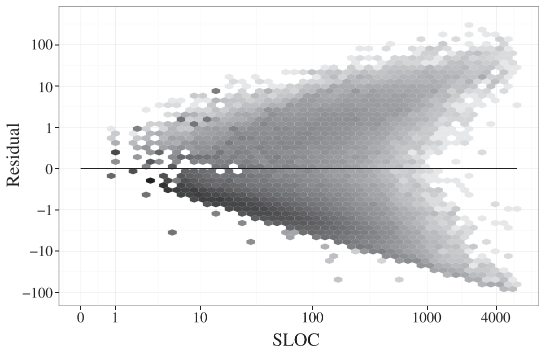
is reminiscent of the apparent ratios between 1st quartile, median, mean, and 3rd quartile of the Java and C corpora in Table III.

As we will discuss in Section 5, the increasing variance in both dimensions causes the largest subroutines to have a large effect on linear correlation strength. To dig further, we did read the code of a number of elements in these long tails (selected using a random number generator). For Java, we read 10 methods out of 1762 with SLOC > 688, and for C, we also read 10 functions out of the 652 with SLOC > 2622. We observed that five out of these 10 methods in Java were clearly generated code and four out of the 10 sampled C functions as well.

We further analyze the strength of the linear correlation after log transform (Hypothesis 5). Figure 11 shows the residual plot of the dashed line shown in the scatter plots. A residual plot



(a) Java



(b) C

Figure 11. Residual plot of the linear regressions after the log transform, where both axes are on a log scale. The grayscale gradient of the hexagons is logarithmic. SLOC, source lines of code.

displays the difference between the prediction and the actual data. For a good model, the error should contain no pattern and have a random distribution around the zero-line. Here, we clearly see the variance in CC increasing as SLOC increases. This further supports results from Table IV, where the prediction error for CC grows with higher SLOC.

This increasing variance we observed is a form of heteroscedasticity. Heteroscedasticity refers to the non-constant variance of the relation between two variables. The Breusch–Pagan test [73] confirmed ($p < 2.20 \times 10^{-16}$) that the relation between CC and SLOC is indeed heteroscedastic for both Java and C. Heteroscedasticity may bias estimated standard errors for the regression parameters [73], making the interpretation of the linear regression potentially error prone.

4.4. *Alternative explanations*

This subsection will explorer alternative explanations to further understand the impact of different choices made by related work (Section 2.2).

4.4.1. CC variant. As discussed in Section 2.1, there is confusion on which AST nodes should be counted for CC. To understand the effect of this confusion on the correlation, we have also calculated the CC without counting the && and || Boolean operators. The CC changed for 1.3 M of the 17.6 M Java methods, of which the CC of 74.2 K methods changed by more than 50%. For C, 1.5 M of the 6.3 M functions had a different CC, of which the CC of 73.3 K functions changed by more than 50%. However, this change has negligible effect on correlation. For Java, the R^2 changed from 0.40 to 0.41, and for C, it stayed at 0.44. Similarly, small effects were observed for other ranges of Tables IV and V.

4.4.2. Aggregation. To investigate Hypothesis 3, we have also aggregated CC and SLOC on file level. This A/B experiment isolates the factor of aggregation. In Table VI, the ‘None’ rows repeat the R^2 before aggregation for Java and C (cf. the first rows in Table V). The ‘File’ rows show the R^2 for the aggregated CC and SLOC before and after the log transform.

Figure 12 shows the hexagonal scatter plots for the aggregation on file level. The two black lines show the linear regression before and after the log transform. The dashed line is the regression after log transform. It can be observed that for larger files, these regressions do not seem to fit the data, that is, smaller files dominate the fitting of the regression line.

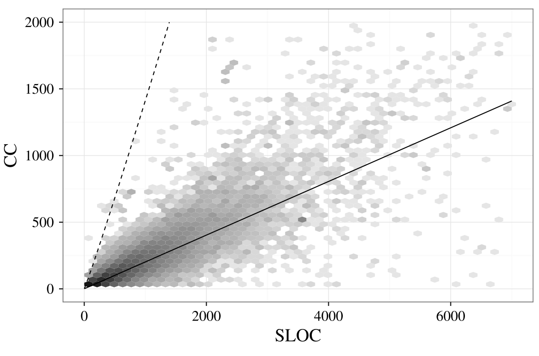
Because the previous experiment includes the confounding factor of header size, we now report on another A/B test to investigate Hypothesis 4. We aggregate the *subroutine values* of CC and SLOC on file level. The ‘$\sum$ Method’ and ‘$\sum$ Function’ rows in Table IV indicate the increase of R^2 both for Java and C.

In Section 4.3, we showed how the non-constant variance (heteroscedasticity) causes the largest subroutines to have a large impact on the correlations. To investigate the difference between file-level (Hypothesis 3) and subroutine-level (Hypothesis 4) aggregation, we also report the effect of removing the largest files on the correlations. Removing the 5‱ largest files from Java (848 files) and C (231 files)—similarly to Section 4.3—improves R^2 to 0.83 (from 0.64) for Java and 0.64 for C (from 0.39).

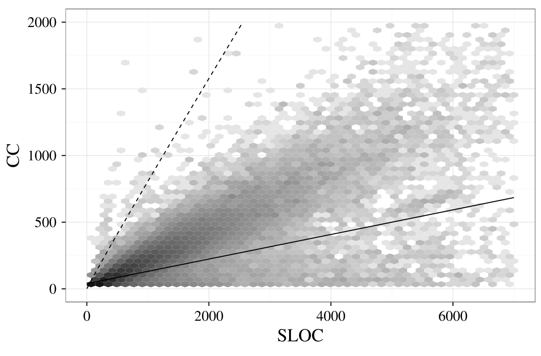
Table VI. Correlations (before and after a log transform) between the aggregated SLOC and CC metrics on a file level (Hypothesis 3) and after summing only the bodies of the subroutines (Hypothesis 4).

Language	Aggregation	R^2	log R^2
Java	None	0.40	0.68
	File	0.64	0.87
	Σ Method	0.73	0.90
C	None	0.44	0.71
	File	0.39	0.84
	Σ Function	0.70	0.90

The first rows per language are a copy of the first rows in Tables V.



(a) Java



(b) C

Figure 12. Scatter plots of source lines of code (SLOC) versus cyclomatic complexity (CC) for Java and C files. The solid and dashed lines are the linear regression before and after the log transform. The grayscale gradient of the hexagons is logarithmic.

Digging further to see what kind of code could have such a large impact, we used a random number generator to sample 10 large files for both corpora (SLOC > 3601 for Java and SLOC > 19934 for C). We then manually inspected the source code in these files. Five out of 10 files were clearly generated code in the Java selection and nine out of 10 in the C selection. Two of these generated C files were the result of a process called 'amalgamation' where the developer includes all hand-written code of a library project into a single file to help C compiler optimization or ease deployment.

4.5. Spearman correlation

Although our main hypothesis is about linear Pearson correlation, we can compute Spearman's correlation to find out if there is a monotone relation. The results are also in Tables IV and V, showing reasonably high ρ values, but decreasing rapidly when we move out of the lower ranges that the distribution skews towards.

This indicates that for the bulk of the data, it is indeed true that a new conditional leads to a new line of code, an unsurprising and much less profound observation than the acceptance or rejection of Hypothesis 1. However, it is still interesting to observe the decline of the Spearman correlation for higher SLOC, which reflects the fact that many different combinations of SLOC and CC are being exercised in the larger methods of the corpus.

5. DISCUSSION

Here, we interpret the results from Section 4. Note that we only have results for Java and C and we sometimes compare these informally to results on different programming languages summarized in Table I.

5.1. Hypothesis 1—strong Pearson correlation

Compared with R^2 between 0.51 and 0.96 [6, 8, 23, 24, 29–31, 34, 35] summarized in Table I, our R^2 of 0.40 and 0.44 are relatively low. This is reason enough to reject the hypothesis: for Java methods and C functions, there is no evidence of a strong linear correlation between SLOC and CC in these large corpora, suggesting that—at least for Java and C—CC measures a different aspect of source code than SLOC, or that other confounding factors are generating enough noise to miss the relation. Here, we focus on related work with the same aggregation level and without log transforms. We conclude that these results, for different programming languages and smaller corpora, do not generalize to our corpora. For higher aggregation levels, see our discussion of Hypothesis 3 in the succeeding text.

The cause of the low R^2 in our data seems to be the high variance of CC over the whole range of SLOC. We observe especially that the variance seems to increase when SLOC increases: the density of control flow statements for larger subroutines is not a constant. This heteroscedasticity is confirmed by the Breusch–Pagan test. Of course, the shape of the distribution influences the results as well, which we investigate while answering Hypothesis 5.

There is no evidence for strong linear correlation between CC and SLOC. Lower R^2 values can be attributed to high variance of CC for the whole range of SLOC.

5.2. Hypothesis 2—no effect of Boolean operators

The results show that the corpora did not contain significant use of the short-circuit Boolean operators. At least, there is not enough support to change the conclusion of Hypothesis 1. We can therefore not reject Hypothesis 2.

Nevertheless, the CC of 8% Java methods and 23% C functions that do use Boolean operators are influenced. It is interesting to note that these subroutines sometimes had very long lines. These subroutines would be missed when counting only SLOC or when ignoring the operators for CC.

What we conclude is that the difference between related work and our results cannot be explained by a different version of CC, because changing it does not affect the correlation. Our recommendation is that for Java and C, the CC computation should include the && and || Boolean operators, because they do measure a part of the control flow graph as discussed in Section 2.

Lack of correlation cannot be explained by including or excluding Boolean operators in the calculation of CC.

5.3. Hypotheses 3 and 4—effect of aggregation (sum)

Related work [5, 7, 9, 22, 25–28, 32, 33, 36–41, 43, 45–49] reported high correlations between CC and SLOC on a larger than methods/functions/subroutines level. For Java, we found similar high correlation after aggregating CC and SLOC on a file level, however not for C. After removing the largest 5‱ files for C, we also do not find better correlations. Hypothesis 3 can therefore not be rejected for Java, but it is rejected for the C corpus. Hence, for the Java corpus, we may conclude that a high R^2 is indeed caused by summing up CC. For the C corpus, we investigated if another influencing factor such as the variance in the header code (Sections 2.3 and 4.4.2) could explain the rejection of Hypothesis 3.

Hypothesis 4 was introduced, therefore, to investigate the impact of the header code (in files) on the correlation values as opposed to summation of the values at the subroutine level. The only difference between Hypotheses 3 and 4 is the inclusion or exclusion of SLOC outside the subroutine bodies for the entire corpus. For Java and C, we both found high correlations after aggregating CC and SLOC on a subroutine level, that is, taking the sum of the CC and SLOC for all subroutines in a file. These observations support Hypothesis 4 (now also for the C corpus) and indicate that the variance of SLOC in the header was indeed a confounding factor for the previous experiment. High correlation between the number of methods and the number of fields reported by Grechanik *et al.* [70] might explain why header size did not have confounding effect for Java. We conclude that Hypothesis 4 is not rejected for both Java and C.

Previously, we rejected Hypothesis 1—a strong Pearson correlation for non-aggregated data. So, we have a strong indication that the related work reporting a high correlation based on a file-level aggregation is likely caused by the aggregation itself rather than a linear relation between SLOC and CC. Because we cannot literally reproduce the data of the related work, this conclusion must remain a conjecture, but the previous experiments do isolate a strong effect of aggregation on our corpora.

In conclusion, the number of subroutines is a factor of system size, and aggregation influences the correlation positively. Similar observation has been made for the relation between SLOC and the number of defects [52]. Therefore, we deem aggregated CC more unnecessary as level of aggregation grows larger (classes, packages, systems). If CC should be aggregated for another (external) reason, more advanced aggregation techniques such as econometric inequality indexes [52–54] should be used rather than sum.

Summing CC and SLOC on a file level could have caused high correlations reported in related work.

5.4. Hypothesis 5—positive effect of the log transform

As reported in related work [6, 43, 46, 47], a log transform indeed increases the R^2 values (from 0.40 to 0.68 for Java and from 0.44 to 0.71 for C). Because of this, we do not reject Hypothesis 5. This finding agrees with the earlier observation on the impact of the log transform on R^2 [58].

However, what does a high Pearson correlation after log transform suggest for the relation between SLOC and CC? Does it have predictive power?

Recall that the Pearson correlation estimates a linear model like this: $\mathrm{CC} = \alpha + \beta \cdot \mathrm{SLOC}$. Hence, if the model after the log transform is $\log_{10}(\mathrm{CC}) = \alpha + \beta \cdot \log_{10}(\mathrm{SLOC})$, then $\mathrm{CC} = 10^{\alpha} \cdot \mathrm{SLOC}^{\beta}$, which implies the nonlinear and monotonic model. Note that the R^2 of 0.68 and 0.71 do not have a natural interpretation in this nonlinear model. Indeed, as recognized in the literature [74, 75], the log-scale results must be retransformed to the original scale, leading to 'a very real danger that the log scale results may provide a very misleading, incomplete, and biased estimate of the impact of covariates on the untransformed scale, which is usually the scale of ultimate interest' [74]. The experiment resulting in a Spearman ρ at 0.80 and 0.83 does confirm the monotonicity as well as the correlation, but it does not help interpreting these results.

Comparing this R^2 after the log transform to the R^2 before transformation is a complex matter; indeed, the literature suggests that the R^2 values are not comparable [57, 58]. In the lower range of SLOC and CC, the effect of the log transform is small; however, as SLOC increases, so does the impact of the transform. Furthermore, the variance of the model after the transform increases much with higher SLOC as well (Figure 11). We conclude that the observations of an R^2 being higher after transform reinforce the conclusion of Hypothesis 1 (there is no strong Pearson correlation) but do not immediately suggest that there exists an exponential relation between SLOC and CC. The variance is too high and not predictable enough.

In combination with aggregation (sum), log transform has lead to the highest R^2 values observed (cf. Table IV). However, the regression lines do not fit the data for larger files (cf. Figure 12). This is caused by the heavy skew of the distributions towards the smaller values.

What we conclude is that the relatively high correlation coefficients after a log transform in literature are reinforced by our own results. These results provide no evidence of CC being redundant to SLOC because the nonlinear model cannot easily be interpreted with accuracy.

A log transform increases the R^2 values between CC and SLOC; however, interpreting the model in terms of the untransformed variables is complex.

5.5. Hypotheses 6 and 7—positive effect of zooming

The final try was to find linear correlation on parts of the data, in order to compensate for the shape of distributions. Our results show that zooming in on tails reduced the correlation, while zooming in on the heads improved it for the 80–100% range. Intuitively, if we remove all elements from the tail of the distributions, then we may achieve the highest R^2 (0.59 for Java and 0.67 for C).

Based on the data, we reject Hypothesis 6 (hypothesizing an effect of the smallest elements), and we do not reject Hypothesis 7 (hypothesizing an effect of a long tail). These results are corroborated in Tables IV and V, showing that the log transform only improves the correlation for the whole range.

We interpret the large effect of tail elements to the increasing variance with high SLOC (heteroscedasticity), rather than label them as 'outliers'. There is no reason to assume the code is strange, erroneous, or false more than the elements in the prefix of the data can be considered strange. The benefit of having the two big corpora is that there are enough elements in the tail to reason about their effect with confidence.

Our analysis, however, does motivate that (depending on the goals of measuring source code) tool vendors may choose to exclude elements from the tail when designing their predictive or qualitative models. Note, however, that even the head of the data suffers from heteroscedasticity, so the same tool vendors should still not assume a linear model between SLOC and CC.

The results for Hypothesis 6 and Hypothesis 7 support our original interpretation for the main Hypothesis 1: CC is not redundant for Java methods or C functions. Nevertheless, the data also show enormous skew towards the smallest subroutines (2 or 3 lines), for which clearly CC offers no additional insight over SLOC. If a Java system consists largely of very small methods, then its inherent complexity is probably represented elsewhere which can be observed using OO specific metrics such as the Chidamber and Kemerer suite [21].

For the larger subroutines, and even the medium-sized subroutines, correlation decreases rapidly. This means that for all but the smallest subroutines, CC is not redundant. For example, looking at the scatter plot in Figure 8 and the box plots in Figure 10, we see that given a Java method of 100, CC has a range between 1 and 40, excluding the rare exceptions. In our Java corpus, there are still 104 K methods larger than or equal to 100. For such larger Java methods, CC can be a useful metric to further discriminate between relatively simple and more complex larger methods. We refer to our previous work [1] and the work of Abran [55] for a discussion on the interpretation of the CC metric on large subroutines.

Large subroutines have a negative influence on the correlations. They are not always generated code; therefore, labeling them as outliers should be performed with care.

5.6. Comparing Java and C

Java and C are different languages. While Java's syntax is strongly influenced by C (and C++), the languages represent different programming paradigms (respectively object-oriented programming and procedural programming). While one could write procedural code in Java (the most common model for C), OO style is encouraged and expected.

In our corpora, C functions are larger and have more control flow than Java methods (Figures 7 and 8). Future work could investigate whether this difference is caused by the difference in programming paradigm and coding idioms or this is caused by another factor such as application domain.

Note that the mode of both SLOC and CC are the same for Java and C. We also observe similar shapes in the scatter plots (Figures 8 and 9): both corpora feature increasingly high variance. We must conclude that although the corpora quantitatively have a different relation between CC and SLOC, qualitatively we come to the same conclusions of a relatively weak linear correlation.

On the one hand, for the C language, we observed that after aggregation to the file level the correlation strength went down. We attributed the cause to the SLOC of C header code (the code outside the function bodies) having high variance. This obscures the relation between SLOC and CC for the C language on the file level, which was confirmed by testing for an increased correlation strength after measuring only the SLOC sum of functions per file. On the other hand, for Java, it appears the header code is not a confounding factor. Again, this is not the point of the current paper, but we conjecture that the stronger encapsulation primitives, which Java offers, bring upon a stronger relation (cohesion) between header code and subroutine bodies.

The differences between C and Java code do not offer additional insight for the relation between SLOC and CC in open-source code, other than an increased external validity of the analysis of Hypothesis 1. Our conclusions hold for both languages.

5.7. Threats to validity

Next to the threats to validity we have identified in the experimental setup (Section 3) and the previous discussion, we further discuss a few other important threats to validity here.

5.7.1. Construct validity. Construct validity pertains to our ability to model the abstract hypothesis using the variables we have measured [76]. We do not believe our study to be subject to construct validity threats because the abstract hypothesis we have tested (Hypothesis 1) has already been formulated in terms of measurable variables (SLOC, CC, and R^2) as opposed to more abstract constructs (e.g., maintainability or development effort).

As to the use of Pearson's coefficient, this was motivated by its common use in related work, which we tried to replicate. Our negative conclusions, meaning we deem the observed R^2 values significantly lower, are subject to the critical examination of the reader.

5.7.2. Internal validity. We have tested the tools we developed for our experiments and compared the output to manually expected results and other free and open-source metric tools. Moreover, to mitigate any unknown issues and to allow for full reproducibility, we have also published both our data and scripts at http://homepages.cwi.nl/~landman/jsep2015/.

To handle the preprocessor statements in C, we have used a heuristic (Section 3). This heuristic filtered away 7% of the code in the corpus. We also filtered all C files with unbalanced braces, which may have been introduced by the aforementioned preprocessor heuristics —not a } for every {. This removed 4 K files (0.50%) from the corpus. There is no reason to expect these filters have introduced a bias for either the SLOC or the CC variables, but without these filters, the corpus would have contained invalid data.

Different from related work [47], we chose not to exclude all .h files (Section 3.4). If we do ignore all .h files, the R^2 for the subroutine level changes from 0.435 to 0.441, that is, both 0.44 when rounded to two significant digits.

5.7.3. External validity. Both our corpora were constructed from open-source software projects containing either Java or C code. Therefore, our results should not be immediately generalized to proprietary software or software written in other programming languages. We should observe that although both languages and their respective corpora are significantly different, we do arrive at similar conclusions regarding our hypotheses. We therefore conjecture that given a comparably large corpus for C-like programming languages (e.g., C++, Pascal, C#), the results should be comparable. A recent study of rank-based correlation between CC and SLOC in Scala GitHub repositories [77] suggests that our results might be valid for Scala as well. While CC adaptations have also been proposed for such languages as Miranda [78] and Prolog [79], those adaptations are quite remote from the original notion of CC as introduced by McCabe [15], and therefore, the relation between CC and SLOC for these languages might be very different.

Moreover, we are aware that the size of the corpus may be a confounding factor and therefore should be investigated [80] and that our study might have been biased by presence of certain accidental data points in our corpora.

Therefore, we performed an additional sensitivity analysis [81] for which the results are reported in the succeeding text. To assess whether the size of the corpus has an important influence on the result, we test whether the strength of the linear correlation between SLOC and CC is similar for randomly selected sub-corpora of half the size. Figure 13 shows the distribution of R^2 values for 1000 random sub-corpora.

The medians of the R^2 values are very close—up to two significant digits—to the R^2 of the full corpora. However, there is a visible spread for the non-transformed variables, before and after aggregation on file level. The log transform has a clearly stabilizing effect because of compression

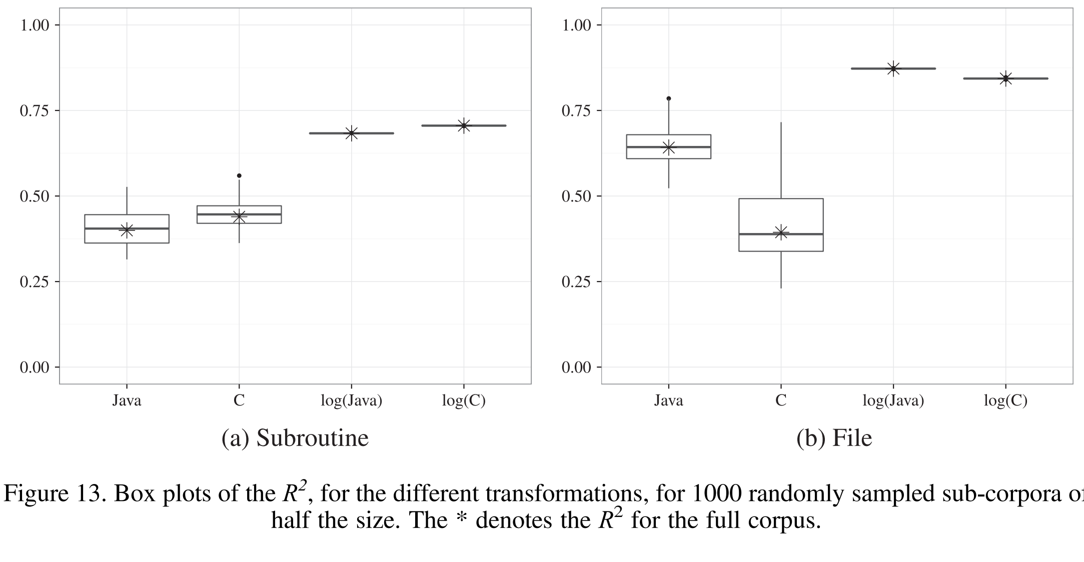


Figure 13. Box plots of the R^2, for the different transformations, for 1000 randomly sampled sub-corpora of half the size. The * denotes the R^2 for the full corpus.

of the tail. Because of the latter observation, we argue once more that the observed effects can be explained by the increasing variance in the tail of the data (cf. Table V). Randomly selected subsets filter a number of elements from the tail, explaining the spread between the 1000 experiments. Moreover, the R^2 in these experiments are not contradicting our previous discussion of the hypotheses.

The previous experiment mitigates the risk of the size factor confounding our observations and conclusions: it can be expected that for random sub-corpora of half the size the correlation strength is the same as for the full corpora.

In contrast, we believe that the large size of the corpora has made it possible to observe the relation between CC and SLOC on arbitrary real-world code with no other known biases.

6. CONCLUSION

The main question of this paper was if CC correlates linearly with SLOC and if that would mean that CC is redundant. In summary, as opposed to the majority of the previous studies, we did not observe a strong linear correlation between CC and OC of Java methods and C functions. Therefore, we do not conclude that CC is redundant with SLOC.

Factually, on our large corpora of Java methods and C functions, we observed (Section 4) the following:

- CC has no strong linear correlation with SLOC on the subroutine level.
- The variance of CC over SLOC increases with higher SLOC.
- Ignoring && and || has no influence on the correlation.
- Aggregating CC and SLOC over files improves the strength of the correlation.
- A log transform improves the strength of the correlation.
- The correlation is lower for larger (SLOC) methods and functions.
- Excluding the largest methods and functions improves the strength of the correlation.
- The largest methods and functions are not just generated code and therefore should not be ignored when studying the relation between SLOC and CC.

From our interpretation of this data (Section 5), we concluded the following:

- CC summed over larger code units measures an aspect of system size rather than internal complexity of subroutines. This largely explains the often reported strong correlation between CC and SLOC in literature.
- Higher variance of CC over SLOC observed in our study as opposed to the related work can be attributed to our choice for much larger corpora, enabling one to observe many more elements.
- The higher correlation after a log transform, supporting results from literature, should not be interpreted as a reason for discarding CC.
- All the linear models suffered from heteroscedasticity, that is, non-constant variance, further complicating their interpretation.

Our work follows the ongoing trend of empirically re-evaluating (or even replicating [82]) earlier software engineering claims (cf. [83, 84]). In particular, we believe that studying big corpora allows to observe features of source code that would otherwise be missed [85].

REFERENCES

1. Vinju JJ, Godfrey MW. What does control flow really look like? Eyeballing the cyclomatic complexity metric. *9th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM)*, IEEE Computer Society, 2012.
2. von Mayrhauser A, Vans AM. Program comprehension during software maintenance and evolution. *IEEE Computer* 1995; **28**(8):44–55.
3. Baggen R, Correia JP, Schill K, Visser J. Standardized code quality benchmarking for improving software maintainability. *Software Quality Journal* 2012; **20**(2):287–307. DOI:10.1007/s11219-011-9144-9.
4. Heitlager I, Kuipers T, Visser J. A practical model for measuring maintainability. *Proceedings of 6th International Conference on Quality of Information and Communications Technology*, 2007; 30–39.

5. Sheppard SB, Curtis B, Milliman P, Borst MA, Love T. First-year results from a research program on human factors in software engineering. *AFIPS Conference Proceedings*, vol. 48, New York, NY, USA, 1979; 1021–1027.
6. Feuer AR, Fowlkes EB. Some results from an empirical study of computer software. *Proceedings of the 4th International Conference on Software Engineering*, ICSE '79, IEEE Press: Piscataway, NJ, USA, 1979; 351–355.
7. Basili VR, Perricone BT. Software errors and complexity: an empirical investigation. *Communications of the ACM* 1984; **27**(1):42–52.
8. Jbara A, Matan A, Feitelson DG. High-MCC functions in the Linux kernel. *Empirical Software Engineering* 2014; **19**(5):1261–1298. DOI:10.1007/s10664-013-9275-7.
9. Fenton N, Ohlsson N. Quantitative analysis of faults and failures in a complex software system. *Software Engineering, IEEE Transactions on Aug* 2000; **26**(8):797–814. DOI:10.1109/32.879815.
10. Landman D, Serebrenik A, Vinju JJ. Empirical analysis of the relationship between CC and SLOC in a large corpus of Java methods. *30th IEEE International Conference on Software Maintenance and Evolution, ICSME 2014*, IEEE, 2014; 221–230, DOI:10.1109/ICSME.2014.44.
11. Linstead E, Bajracharya SK, Ngo TC, Rigor P, Lopes CV, Baldi P. Sourcerer: mining and searching internet-scale software repositories. *Data Mining and Knowledge Discovery* 2009; **18**(2):300–336. DOI:10.1007/s10618-008-0118-x.
12. Shepperd M. A critique of cyclomatic complexity as a software metric. *Software Engineering Journal Mar* 1988; **3**(2):30–36. DOI:10.1049/sej.1988.0003.
13. Curtis B, Carleton A. Seven plus ± two software measurement conundrums. *Proceedings of the Second International Software Metrics Symposium*, 1994; 96–105.
14. Conte SD, Dunsmore HE, Shen VY. Software Engineering Metrics and Models. Benjamin-Cummings Publishing Co., Inc.: Redwood City, CA, USA, 1986.
15. McCabe TJ. A complexity measure. *IEEE Transactions Software Engineering* 1976; **2**(4):308–320.
16. Lincke R, Lundberg J, Löwe W. Comparing software metrics tools. *Proceedings of the 2008 International Symposium on Software Testing and Analysis*, ISSTA '08, ACM: New York, NY, USA, 2008; 131–142, DOI:10.1145/1390630.1390648.
17. Myers GJ. An extension to the cyclomatic measure of program complexity. *SIGPLAN Notices October* 1977; **12**(10):61–64. DOI:10.1145/954627.954633.
18. Wohlin C. Guidelines for snowballing in systematic literature studies and a replication in software engineering. *18th International Conference on Evaluation and Assessment in Software Engineering, EASE*, ACM, 2014; 38:1–38:10. DOI:10.1145/2601248.2601268.
19. da Mota Silveira Neto PA, Engström E, de Carmo Machado I, de Almeida ES. On the reliability of mapping studies in software engineering. *Journal of Systems and Software* 2013; **86**(10):2594–2610. DOI:10.1016/j.jss.2013.04.076.
20. Kitchenham B, Charters S. Guidelines for performing systematic literature reviews in software engineering. *Technical Report EBSE 2007-001*, Keele University and Durham University Joint Report 2007.
21. Chidamber SR, Kemerer CF. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering Jun* 1994; **20**(6):476–493. DOI:10.1109/32.295895.
22. Hindle A, Godfrey M, Holt R. Reading beside the lines: indentation as a proxy for complexity metric. *IEEE International Conference on Program Comprehension*, 2008; 133–142, DOI:10.1109/ICPC.2008.13.
23. Capiluppi A, Fernandez-Ramil J. A model to predict anti-regressive effort in open source software. *IEEE International Conference on Software Maintenance, 2007*, ICSM 2007, 2007; 194–203, DOI:10.1109/ICSM.2007.4362632.
24. Malhotra R, Singh Y. On the applicability of machine learning techniques for object oriented software fault prediction. *Software Engineering: An International Journal* 2011; **1**:24–37.
25. Yu L, Mishra A. An empirical study of Lehman's law on software quality evolution. *International Journal of Software & Informatics* 2013; **7**(3):469–481.
26. Posnett D, Filkov V, Devanbu P. Ecological inference in empirical software engineering. *26th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2011; 362–371. DOI: 10.1109/ASE.2011.6100074.
27. Vasilescu B, Serebrenik A, van den Brand MGJ. By no means: a study on aggregating software metrics. *2nd International Workshop on Emerging Trends in Software Metrics*, WETSoM, ACM, 2011; 23–26.
28. Vasilescu B, Serebrenik A, van den Brand MGJ. You can't control the unfamiliar: a study on the relations between aggregation techniques for software metrics. *IEEE 27th International Conference on Software Maintenance, ICSM2011*, 2011; 313–322. DOI:10.1109/ICSM.2011.6080798.
29. Mordal K, Anquetil N, Laval J, Serebrenik A, Vasilescu B, Ducasse S. Software quality metrics aggregation in industry. *Journal of Software: Evolution and Process* 2013; **25**(10):1117–1135. DOI:10.1002/smr.1558.
30. Abran A. Cyclomatic Complexity Number: Analysis of Its Design. Software Metrics and Software Metrology. chap. 6, Clements A (ed.). Wiley-IEEE Computer Society Pr, 2010; 131–143.
31. Jay G, Hale JE, Smith RK, Hale DP, Kraft NA, Ward C. Cyclomatic complexity and lines of code: empirical evidence of a stable linear relationship. *Journal of Software Engineering and Applications* 2009; **2**(3):137–143.
32. Herraiz I, Gonzalez-Barahona JM, Robles G. Towards a theoretical model for software growth. *Proceedings of the Fourth International Workshop on Mining Software Repositories*, MSR '07, IEEE Computer Society: Washington, DC, USA, 2007; 21:1–21:8, DOI:10.1109/MSR.2007.31.
33. Herraiz I, Hassan AE. Beyond lines of code: Do we need more complexity metrics? Making Software What Really Works, and Why We Believe It. chap. 8, Oram A, Wilson G (eds.). O'Reilly Media, 2010; 126–141.
34. Sheskin DJ. Handbook of Parametric and Nonparametric Statistical Procedures, 4 edn. Chapman & Hall/CRC, 2007.

35. Kvålseth TO. Cautionary note about R^2. *The American Statistician* 1985; **39**(4):279–285.
36. Loehle C. Proper statistical treatment of species-area data. *Oikos* 1990; **57**(1):143–145.
37. Troster J, Tian J. Measurement and defect modeling for a legacy software system. *Annals of Software Engineering* 1995; **1**:95–118. DOI:10.1007/BF02249047.
38. Henry S, Selig C. Predicting source-code complexity at the design stage. *Software, IEEE March* 1990; **7**(2):36–44. DOI:10.1109/52.50772.
39. van der Meulen MJ, Revilla MA. Correlations between internal software metrics and software dependability in a large population of small C/C++ programs. *Proceedings of the The 18th IEEE International Symposium on Software Reliability*, ISSRE '07, IEEE Computer Society: Washington, DC, USA, 2007; 203–208, DOI:10.1109/ISSRE.2007.6.
40. Landman D. A curated Corpus of Java source code based on Sourcerer (2015). Available from: http://persistent-identifier.org/?identifier=urn:nbn:nl:ui:18-23357 2015. [25 February 2015].
41. GitHub. Linguist: language savant. Available from: https://github.com/github/linguist 2015. [10 February 2015].
42. Landman D. A large corpus of C source code based on Gentoo packages. Available from: http://persistent-identifier.org/?identifier=urn:nbn:nl:ui:18-23154 2015. [10 February 2015].
43. Basten B, Hills M, Klint P, Landman D, Shahi A, Steindorfer M, Vinju J. M^3: a general model for code analytics in Rascal. *Proceedings of the first International Workshop on Software Analytics, SWAN*, 2015. [To appear].
44. Klint P, van der Storm T, Vinju JJ. RASCAL: a domain specific language for source code analysis and manipulation. *Proceedings of the 9th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM)*, IEEE, 2009; 168–177. DOI:10.1109/SCAM.2009.28.
45. Carr DB, Littlefield RJ, Nicholson WL, Littlefield JS. Scatterplot matrix techniques for large N. *Journal of the American Statistical Association* 1987; **82**(398):424–436.
46. Pearson K. Note on regression and inheritance in the case of two parents. *Proceedings of the Royal Society of London* 1895; **58**:240–242.
47. Edgell SE, Noon SM. Effect of violation of normality on the *t* test of the correlation coefficient. *Psychological Bulletin* 1984; **95**(3):576–583.
48. Spearman C. The proof and measurement of association between two things. *The American Journal of Psychology* 1904; **15**(1):72–101.
49. Joanes DN, Gill CA. Comparing measures of sample skewness and kurtosis. *Journal of the Royal Statistical Society: Series D (The Statistician)* 1998; **47**(1):183–189. DOI:10.1111/1467-9884.00122.
50. Grechanik M, McMillan C, DeFerrari L, Comi M, Crespi S, Poshyvanyk D, Fu C, Xie Q, Ghezzi C. An empirical investigation into a large-scale Java open source code repository. *Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, ACM: New York, NY, USA, 2010; 11:1–11:10, DOI:10.1145/1852786.1852801.
51. Munson JC, Kohshgoftaar TM. Measurement of data structure complexity. *Journal of Systems and Software* 1993; **20**(3):217–225. DOI:10.1016/0164-1212(93)90065-6.
52. Chambers JM, William S Cleveland BK, Tukey PA. Comparing data distributions. Graphical Methods for Data Analysis. chap. 2, Chapman and Hall: New York, 1983.
53. Breusch T, Pagan A. A simple test for heteroscedasticity and random coefficient variation. *Econometrica* Sep 1979; **47**(5):1287–1294.
54. Curtis B, Sheppard SB, Milliman P. Third time charm: stronger prediction of programmer performance by software complexity metrics. *Proceedings of the 4th International Conference on Software Engineering*, ICSE '79, IEEE Press: Piscataway, NJ, USA, 1979; 356–360.
55. Woodward MR, Hennell MA, Hedley D. A measure of control flow complexity in program text. *IEEE Transactions on Software Engineering Jan* 1979; **5**(1):45–50. DOI:10.1109/TSE.1979.226497.
56. Lind RK, Vairavan K. An experimental investigation of software metrics and their relationship to software development effort. *IEEE Transactions on Software Engineering May* 1989; **15**(5):649–653. DOI:10.1109/32.24715.
57. Lewis J, Henry S. A methodology for integrating maintainability using software metrics. *Software Maintenance, 1989., Proceedings., Conference on*, 1989; 32–39, DOI:10.1109/ICSM.1989.65191.
58. Gorla N, Benander A, Benander BA. Debugging effort estimation using software metrics. *IEEE Transactions on Software Engineering* 1990; **16**(2):223–231.
59. O'Neal MB. An empirical study of three common software complexity measures. *Proceedings of the 1993 ACM/SIGAPP Symposium on Applied Computing: States of the Art and Practice*, SAC '93, ACM: New York, NY, USA, 1993; 203–207, DOI:10.1145/162754.162867.
60. Kemerer CF, Slaughter SA. Determinants of software maintenance profiles: an empirical investigation. *Journal of Software Maintenance* Jul 1997; **9**(4):235–251.
61. Paige M. A metric for software test planning. *Conference Proceedings of COMPSAC* 1980; **80**:499–504.
62. Sunohara T, Takano A, Uehara K, Ohkawa T. Program complexity measure for software development management. *Proceedings of the 5th International Conference on Software Engineering*, ICSE '81, IEEE Press: Piscataway, NJ, USA, 1981; 100–106.
63. Li H, Cheung W. An empirical study of software metrics. *IEEE Transactions on Software Engineering June* 1987; **SE-13**(6):697–708. DOI:10.1109/TSE.1987.233475.

64. Kitchenham B, Pickard L. Towards a constructive quality model. part 2: Statistical techniques for modelling software quality in the esprit request project. *Software Engineering Journal July* 1987; **2**(4):114–126. DOI:10.1049/sej.1987.0015.
65. Gill GK, Kemerer CF. Cyclomatic complexity density and software maintenance productivity. *IEEE Transactions on Software Engineering Dec* 1991; **17**(12):1284–1288. DOI:10.1109/32.106988.
66. Graves T, Karr A, Marron J, Siy H. Predicting fault incidence using software change history. *Software Engineering, IEEE Transactions on Jul* 2000; **26**(7):653–661. DOI:10.1109/32.859533.
67. Succi G, Benedicenti L, Vernazza T. Analysis of the effects of software reuse on customer satisfaction in an RPG environment. *IEEE Transactions on Software Engineering May* 2001; **27**(5):473–479. DOI:10.1109/32.922717.
68. El Emam K, Benlarbi S, Goel N, Rai SN. The confounding effect of class size on the validity of object-oriented metrics. *IEEE Transactions on Software Engineering* 2001; **27**(7):630–650.
69. Martin C, Pasquier J, Yanez C, Tornes A. Software development effort estimation using fuzzy logic: a case study. *Sixth Mexican International Conference on Computer Science, 2005.*, ENC 2005, 2005; 113–120. DOI:10.1109/ENC.2005.47.
70. Schneidewind N. Software reliability engineering process. *Innovations in Systems and Software Engineering* 2006; **2**(3-4):179–190. DOI:10.1007/s11334-006-0007-7.
71. Bianco M, Kaneider D, Sillitti A, Succi G. Fault-proneness estimation and java migration: a preliminary case study. *Proceedings of the Software Services Semantic Technologies Conference (S3T 2009)*, 2009; 124–131.
72. Ma YT, He KQ, Li B, Liu J, Zhou XY. A hybrid set of complexity metrics for large-scale object-oriented software systems. *Journal of Computer Science and Technology* 2010; **25**(6):1184–1201. DOI:10.1007/s11390-010-9398-x.
73. Tashtoush Y, Al-Maolegi M, Arkok B. The correlation among software complexity metrics with case study. *International Journal of Advanced Computer Research* 2014; **4**(2):414–419.
74. Manning WG. The logged dependent variable, heteroscedasticity, and the retransformation problem. *Journal of Health Economics* 1998; **17**(3):283–295. DOI:10.1016/S0167-6296(98)00025-3.
75. Feng C, Wang H, Lu N, Tu XM. Log transformation: application and interpretation in biomedical research. *Statistics in Medicine* 2013; **32**(2):230–239. DOI:10.1002/sim.5486.
76. Perry DE, Porter AA, Votta LG. Empirical studies of software engineering: a roadmap. *Proceedings of the Conference on The Future of Software Engineering*, ICSE '00, ACM: New York, NY, USA, 2000; 345–355. DOI:10.1145/336512.336586.
77. Coleman R, Johnson MA. A study of Scala repositories on GitHub. *International Journal of Advanced Computer Science and Applications* 2014; **5**(7):141–148.
78. van den Berg KG. Software measurement and functional programming. PhD Thesis, University of Twente, Enschede, the Netherlands, Enschede June 1995.
79. Moores TT. Applying complexity measures to rule-based prolog programs. *Journal of Systems and Software* 1998; **44**(1):45–52. DOI:10.1016/S0164-1212(98)10042-0.
80. Rahman F, Posnett D, Herraiz I, Devanbu P. Sample size vs. bias in defect prediction. *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, ACM: New York, NY, USA, 2013; 147–157. DOI:10.1145/2491411.2491418.
81. Saltelli A, Tarantola S, Campolongo F. Sensitivity analysis as an ingredient of modeling. *Statistical Science* 2000; **15**(4):377–395.
82. Shull FJ, Carver JC, Vegas S, Juristo N. The role of replications in empirical software engineering. *Empirical Software Engineering* 2008; **13**(2):211–218. DOI:10.1007/s10664-008-9060-1.
83. Khomh F, Adams B, Dhaliwal T, Zou Y. Understanding the impact of rapid releases on software quality. *Empirical Software Engineering* 2014; **20**(2):336–373. DOI:10.1007/s10664-014-9308-x.
84. Ray B, Posnett D, Filkov V, Devanbu P. A large scale study of programming languages and code quality in Github. *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ACM: New York, NY, USA, 2014; 155–165, DOI:10.1145/2635868.2635922.
85. Siegmund J, Siegmund N, Appel S. Views on internal and external validity in empirical software engineering. *37th International Conference on Software Engineering*, ACM, 2015.