

Comparing Quality Metrics for Cloned and non cloned Java Methods: A Large Scale Empirical Study

Vaibhav Saini*, Hitesh Sajjani[†] and Cristina Lopes*

*Dept. of Information and Computer Science, University of California Irvine
{vpsaini, hsajjani, lopes}@uci.edu

[†]Microsoft, Inc.

Abstract—In this paper, we conduct a large scale statistical study to explore if there exists any difference between the quality of cloned methods and non cloned methods. The dataset consists of 4,421 open source Java projects containing 644,830 cloned and 842,052 non cloned methods. The study uses 27 software metrics as a proxy for quality, spanning across complexity, modularity, and documentation (code-comments) categories. We did not find any statistically significant difference ($p < 0.05$, $r > 0.1$) between the quality of cloned and non cloned methods for most of the metrics, except for 3 metrics. We, however, found that the cloned methods are on an average 20% smaller than the non cloned methods.

Index Terms—Code Clones, Quality Metrics, Open Source Software

I. INTRODUCTION

Reusing code via copy-and-paste, with or without modifications, has become a common behavior in software engineering [1], exacerbated by the online availability of samples of code provided by individuals or communities via blogs, online forums, or even specialized web sites. This form of code reuse often leads to duplication of code, also known as code cloning.

Traditionally, code cloning has been presented as a design error, or more broadly as a *code-smell* [2], [3]. It has been argued that code cloning adversely impacts software by forcing the tracking of all pieces of code when a certain fault is known to exist, or a new behavior is required, a task that can easily be error prone. Consequently, considerable effort has been put into clone detection [4]–[12], clone removal [13], [14] and clone refactoring [15].

Despite the disadvantages of code cloning, they also leverage the difficulty that software development typically implies. They speed-up development and improve developer productivity [16], allow the reuse of tested code, or allow a better separation of concerns [16], [17].

These two different but defensible opinions raise the question of whether code cloning should be avoided, eliminated, tolerated or even recommended, and under what situations. The interest and the lack of knowledge on the topic led to a call to conduct more empirical studies [18] about code clones. Koschke has presented several important open issues [19], one of which being the relation of clones to quality attributes. To assess the impact of clones on defect occurrence of software products, Rahman et al. conducted a study on 4 projects. They

did not find any evidence that cloning is harmful [20]. In another study conducted on 36 Java projects, Sajjani et al. found that cloned methods have lesser defect density than that of the non cloned methods [21].

Building on this series of studies, we conduct a statistical study to explore the relationship between code clones and 27 software quality metrics on 4,421 Java projects, totaling 1,486,882 methods. As per our knowledge, this is the largest study on code cloning so far.

A. Definitions and Thresholds

We consider a method to be a clone of another, if there exists a syntactically similar method in the same project – we focus on intra-project clones only. The literature defines three types of syntactically similar clones, namely [18], [22]:

- Type-1: Identical code fragments, except for differences in white-space, layout and comments.
- Type-2: Identical code fragments, except for differences in identifier names and literal values, in addition to Type-1 clone differences.
- Type-3: Syntactically similar code fragments that differ at the statement level. The fragments have statements added, modified and/or removed with respect to each other, in addition to Type-1 and Type-2 clone differences.

In this study we include all of these types of clones.

In the statistical analysis, we conduct Wilcoxon Rank-Sum tests or use Linear Regression Models, and we consider *significant difference* to be a combination of the P-value and the effect size of the tests. Specifically, we are looking for P-values smaller than 0.05 (denoted $p < 0.05$) and the effect-size given by Cohen's d or Hedge's g greater than 0.1 (denoted $r > 0.1$).

B. Research Questions

We seek answers to the following research questions:

Research Question 1: *In a project, are cloned methods more or less complex than non cloned methods?*

Unnecessary complexity has obvious disadvantages on a software solution and negatively affects most of its internal characteristics. If cloned code has higher code complexity than non cloned code, then the advantages that code cloning might introduce should be carefully balanced with the negative

impact it has on maintainability, cost-efficiency, security and so on. We use 19 software quality metrics that address code complexity.

Research Question 2: *In a project, are cloned methods more or less modular than non cloned methods?*

We define modularity as the ability of a certain piece of code to independently and modularly represent a specific functionality or semantic property.

If cloned code is found to be less modular than non cloned code, this suggests code cloning should be avoided. Modular artifacts are easier to maintain as their understanding does not depend on their contextual location in the source code.

We compute modularity using 5 *Modularity* metrics.

Research Question 3: *In a project, are cloned methods more or less documented than non cloned methods?*

The importance of well-documented source code is well known, but it is of further relevance in cloning code as programmers are expected to be more unfamiliar with code they simply copy-and-paste. A finding that relates code clones with minimal code comments, or simply with their inexistence, will translate into a software artifact that is hard to understand and debug, comparing to a counterpart where the programmer developed the totality of the source code.

We note that we are only looking at the quantitative aspect of *Documentation* and are not analyzing the documentation qualitatively.

C. Findings

Overall we found no evidence that the quality metrics of cloned methods are significantly different than those of the non cloned methods for the vast majority of the metrics. The metric that showed the most significant difference was size, measured here as the number of statements: on the dataset of 644,830 cloned and 842,052 non cloned methods, we found that cloned methods are 20% smaller than the non cloned methods.

A small fraction of the projects show significant difference in the quality of cloned and non cloned methods, even after controlled for size. Specifically, for 3 metrics out of the total of 27 approximately 30% projects contained cloned methods with significantly different values of the quality metric than non cloned methods. For all the other 24 metrics, only a very small number of projects, or in some cases no projects at all, contained significant differences between clones and non clones.

Outline: The remainder of the paper is organized as follows. Section II discusses related work; Section III explains the study design and introduces the dataset, the clone detection technique and the software quality metrics used in the study; Section IV presents and discusses the results; the limitations of this study are discussed in Section V; finally, we summarize our findings and discuss future work in Section VI.

II. RELATED WORK

Code cloning and its impact has a long history as a topic of inquiry in the research community. In this section, we draw comparisons with existing research in this direction.

Traditionally, researchers have discussed many negative effects of code cloning [3], [23]–[27], however, the widespread presence of clones have motivated researchers to dig deeper and understand the usage scenarios.

Kapser et al. presented eleven patterns by examining clones in two projects [16]. They found out that not all usage patterns have negative consequence and some may even have positive consequence on quality.

Ossher et al. looked at circumstances of file cloning in open source Java projects and classified the cloning scenarios into good, bad and ugly [28]. These scenarios included good use-cases like extension of Java classes and popular third-party libraries, both large and small. They also found ugly cases where a number of projects occur in multiple online repositories, or have been forked, or were copied and divided into multiple sub projects. From a software engineering standpoint, some of these situations are more legitimate than others.

Kim et al. studied clone evolution in two open source projects. They found that most of the clone pairs are short lived and about 72% of the clone pairs diverge within 8 commits in the code repository [17]. They found that several clones exist by design and cannot be refactored because of the limitation of programming language or it would require a design change. Similarly, de Wit et al. proposed CLONEBOARD, an Eclipse plug-in implementation to track live changes to clones and offering several resolution strategies for inconsistently modified clones [29]. They conducted a user study and found that developers see the added value of the tool but have strict requirements with respect to its usability.

Cordy analyzed clones and intentions behind cloning of a financial institution system and argued that external business factors may facilitate cloning [30]. He mentioned that financial institutions avoid situations that can break the existing code under any circumstances. Abstractions might introduce dependencies and modifying such abstractions induces the risk of breaking existing code.

Cloning minimizes this risk as code is maintained and modified separately localizing the risk of errors to a single module. Similarly, Rajapakse et al. found that reducing duplication in a web application only had negative effects on the modifiability of an application [31]. They noted that after significantly reducing the size of the source code, a single change required testing of a vastly larger portion of the project.

Rahman et al. investigate the effect of cloning on defect proneness on four open source projects [20]. They looked at the buggy changes and explored their relationship with cloning. They did not find evidence that cloned code is riskier than non cloned code. This finding is also supported by Sajjani et al. [21], where they conducted an empirical study comparing the presence of bug patterns in cloned and non cloned code on 36 open source Java projects. Surprisingly, they found that the defect density of cloned code is less than 2 times than that of the non cloned code.

Brutnik et al. use clone detection techniques in a novel way to find cross-cutting concerns in the code [32]. They manually identify five specific cross-cutting concerns in an industrial C

project and analyze to what extent clone detection is capable of finding them. The initial results were positive.

Thus, evaluating the positive and negative impacts of cloning has been a continuous balancing act. Like some of the recent studies, our study also presents one more piece of evidence that cloning may not be that bad after all. However, our study differs from similar studies in the literature in several aspects.

We explore the relationship between a set of 27 well known software metrics across three categories and code clones, and compare and contrast the nature of this relationship in non cloned code. To the best of our knowledge, no other study has investigated code cloning using such a comprehensive set of metrics. These metrics are a proactive way of looking at the patterns which are potential threats for the code base. Other studies have looked at the relationship of bugs with cloning by mining the version control systems to find out the bugs associated with clones. While such reported bugs provide a measure of external quality of the overall project, researchers have identified several issues with respect to completeness and correctness of such data. Moreover, like all complex problems, the issue of code cloning being bad or not will only be fully understood by looking at it from several angles and with several methodologies. Moreover, several studies have also evaluated the effectiveness of software quality metric systems and found it to be a useful indicator of code quality [33]–[35].

We conduct this study on 4,421 open source Java projects. Our study is the first to perform clone-quality analysis at this level. This scale not only helps us to instill more confidence in the findings of the study, but also helps to discover and hence account for confounding factors that might impact the validity of the study. The findings of this study not only help in understanding the nature of code clones, but also help in identifying hot spots which are more likely to be re-used in the future.

III. STUDY DESIGN

As shown in Figure 1, the overall design of our study is a two step process. with both steps being merged in a model from which we derive our results. In the first step, we compute the software quality metrics for every method in each project. This step gives us a map to associate methods with their software quality metric values. In the second step, we use a clone detection tool to identify all the intra-project clones present in each project, which results in a map that contains the information of whether a method has a clone or not.

Combining the result of the previous two steps, we have, for each method in a project:

- 1) Whether the method has a clone or not; and
- 2) Software quality metric values for the method

After these steps we had all the information needed to start a statistical analysis and answer the research questions. In the remainder of this section we describe each aspect of our study design in detail.

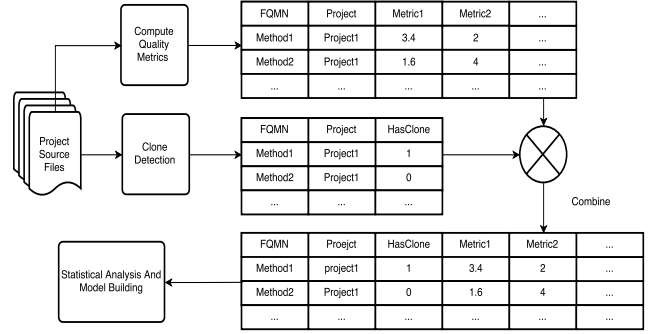


Fig. 1: Study Design
(FQMN is short for Fully Qualified Method Name).

A. Dataset

The dataset used in this study consists of 4,421 Java projects hosted by Maven [36]. The comprehensive list of projects with their version information can be found at <http://mondego.ics.uci.edu/projects/clonemetrics/>. These projects are of varied size and span across various domains including but not limited to search and database projects, server projects, distributed systems, machine learning and natural language processing libraries, and network systems. The large variety of projects was chosen to avoid any bias on our results towards a certain project characteristic, as they represent a wide range of sizes and domains.

Dataset Curation:

We downloaded the entire Maven repository available at UCI Source Code Data Sets [37]. We then selected the top versions of all the projects that have their source files with them. This gave us 18,852 projects. We observed that there were many empty projects in this dataset; so in order to remove very small or empty projects we filtered out all the projects which have less than 100 Java statements. This resulted into a total of 12,872 projects. For every method in each of these projects, we computed clone detection and software quality metrics. As a final filtration step, we rejected the projects that have less than or equal to 10 clone or 10 non cloned methods, resulting into 4,221 projects. We applied this filter for two reasons:

- In order to do a meaningful empirical study of cloned and non cloned methods we wanted to have only those projects which exhibit some extent of cloning; and
- The filter made sure that we consider only the projects that have at least 20 methods (at least 10 non cloned plus 10 cloned methods) in them. Smaller projects may not have many methods/features to clone from. Moreover, a large number of such projects can negatively affect the study.

The top most plot on Figure 2 describes the size distribution of the projects in our dataset. The X-axis represents the binned number of Java statements (NOS). The Y-axis represents the percentage of projects.

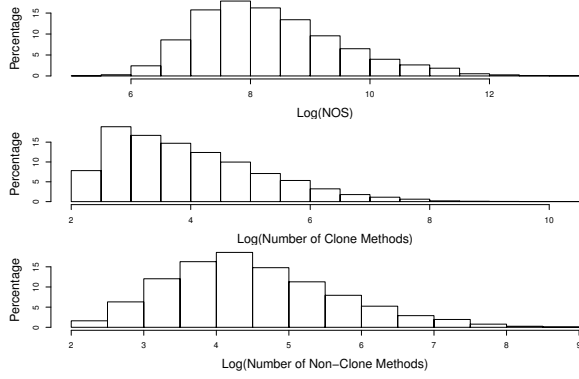


Fig. 2: Size distribution of projects. Top, the number of Java Statements (NOS). Middle, the binned number of cloned methods. Bottom, the binned number of non cloned methods in log scale. The Y-axis shows the percentage of projects. All charts are in log scale.

B. Code Cloning

We use our own scalable token-based clone detection tool, SourcererCC [38].

SourcererCC was developed with scalability as a main concern, and uses a heuristic-optimized index of tokens to significantly reduce the number of code-block comparisons needed to detect the clones, as well as the number of required token comparisons needed to judge a potential clone. Testing has proved the tool to scale to a large repository with 250M lines of code in a single workstation (3Ghz i7 CPU, 12Gb RAM), and the values of precision can reach 86% and recall 93%¹.

Sourcerer-CC is capable of detecting Type 3 method level clones.

We chose the following configuration to run the SourcererCC: threshold value for similarity was set to 70%, meaning methods with proximity '70% or more' were reported by the tool. Methods with less than 25 tokens were ignored, in order to ignore empty methods, stubs, method contracts defined in interfaces, abstract classes and other interferences. At this configuration, SourcererCC has demonstrated to have a precision of 86% and recall of 90% [38].

In total we found 644,830 cloned and 842,052 non cloned methods in our dataset of 4,221 projects. Figure 2 shows the distribution of cloned methods in the projects (middle plot). The X-axis shows the binned number of cloned methods in log scale. The Y-axis shows the percentage of projects. The bottom plot in the same figure shows the similar information but for non cloned methods.

The median size, measured in lines of code (NLOC), for cloned and non cloned methods is 11 and 14 respectively. The median size, measured in number of statements (NOS), for clones and non cloned methods is 8 and 10 respectively (in Section III-C we provide details about these metrics).

¹Specific hardware details and testing conditions are in the original article. The value of recall varies under certain conditions, maximum values are provided here.

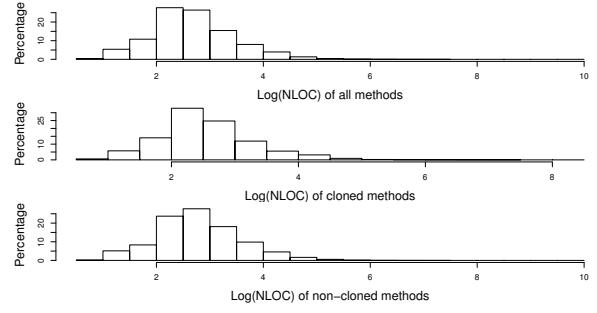


Fig. 3: Distribution of methods measured using number of lines of code metric.

Figure 3 shows the size distribution of all methods (top 2); cloned methods (middle 2); and non cloned methods (bottom 2). The X-axes of the three histograms, show the binned size of methods measured in NLOC in log scale; The Y-Axis shows the number of methods in each bin.

Precision: In order to measure the precision of SourcererCC on our dataset, We randomly selected 401 clone pairs for manual inspection. This is a statistically significant sample with a 95% confidence level and a $\pm 5\%$ confidence interval. The sample set was inspected by 3 judges to figure out the true and false positives. All of the judges have 7+ years of open source software development experience; one has 5+ years of industrial software development experience; and two of them have 3+ years of industrial software development experience. In the cases where there were conflicts, we resolved them by voting. We note that one of the judges is also an author of this paper.

Judge 1 found all 401 as true positives; Judge 2 found 394 as true positives and 7 as false positives; and Judge 3 found 395 as true positives and 6 as false positive. After voting, only one clone-pair was declared as a false positive and the rest 400 were deemed true positives, resulting into high value of 99.7% precision for SourcererCC.

C. Software Quality Metrics

All the metrics used were calculated at the method-level using version 6 of the JHawk tool [39]. JHawk has been widely used in academic studies on Java metrics [40]–[52].

To evaluate quality of the methods, we use the totality of the 27 software quality metrics supported by JHawk, and that can be seen in Table I. We divide the metrics into 3 groups: (i) metrics assessing code complexity; and (ii) metrics assessing modularity properties; and (iii) metrics assessing how well the code is documented. Next we describe each of these categories:

Code Complexity Metrics. These metrics capture the complexity of the code itself. We used metrics as they are the standard technique for analyzing software. For example, a set of metrics used is derived from the metrics included in SQO-OSS, specifically designed for evaluating the quality of open source projects. SQO-OSS is composed of well-established

Category		Name	Description
Code Complexity	↓	COMP	McCabes cyclomatic complexity
	↓	NOA	Number of arguments
	↓	VDEC	Number of variables declared
	↓	VREF	Number of variables referenced
	↓	NOS	Number of statements
	↓	NEXP	Number of expressions
	↓	MDN	Method, Maximum depth of nesting
	↓	HLTH	Halstead length of method
	↓	HVOC	Halstead vocabulary of method
	↓	HVOL	Halstead volume
	↓	HDIF	Halstead difficulty to implement a method
	↓	HEFF	Halstead effort to implement a method
	↓	TDN	Total depth of nesting
	↓	CAST	Number of class casts
	↓	LOOP	Number of loops (for,while)
	↓	NOPR	Total number of operators
	↓	NAND	Total number of operands
	↓	HBUG	Halstead prediction of number of bugs
	↓	NLOC	Number of lines of code
Modularity	↓	CREF	Number of classes referenced
	↓	XMET	External methods called by the method
	↓	LMET	Local methods called by the method
	↓	EXCR	Number of exceptions referenced by the method
	↓	EXCT	Number of exceptions thrown by the method
Documentation	↑	MOD	Number of modifiers
	↑	NOC	Number of comments
	↑	NOCL	Number of comment lines

TABLE I: Software Quality Metrics

and validated software quality metrics, which can be computed either from source code or from surrounding community data.

Moreover, Nagappan et al. conducted a study where many software metrics including but not limiting to *number of executable lines* in a function, *number of parameters* in a function, *number of methods called* by a function or *number of global variables written/read* in a function etc., were shown to correlate with the post-release defects of software [53]. Several studies have also evaluated the effectiveness of software metric projects and found it to be a useful indicator of code quality [33]–[35].

- *COMP* measures the Cyclomatic complexity of a method. It is the standard metric introduced by McCabe in 1976 [54].
- *NOA* measures the total number of arguments expected by a method’s signature.
- *VDEC* is the total number of variables declared in a method.
- *VREF* counts the total number of variables referenced in a method. If a method has a large number of arguments, it declares a large number of variables, or it references a large number of variables then, it may be a sign that the method is trying to do too much. It also means that the changeability, tendency to change, of the method is high.
- *NOS* measures the number of Java statements in a method. A statement forms a complete unit of execution and are usually terminated with a semicolon (;). A statement can be spread over multiple lines; for example one can decide to start the curly brace of an if statement either from the current line or from the next line.
- *NLOC* measures the total number of lines of code in a method.
- *NEXP* measures the total number of expressions in a method. Oracle notes that “An expression is a construct made up of variables, operators, and method invocations, which are constructed according to the syntax of the lan-

guage, that evaluates to a single value”. A higher value of number of expressions per statement may indicate a complex method.

- *MDN* is the measure of maximum depth of nesting in a method. A higher depth of nesting in a method indicates higher complexity of the method.
- *NOPR* and *NAND* measure the total number of operators (arithmetical operators, method names) and the total number of operands (variables, numeric and string constants) used in a method respectively.
- *HLTH*, Halstead length [55], is the most basic of the Halstead metrics, which simply is the sum of *NOPR* and *NAND*.
- *HVOC*, Halstead vocabulary is computed as the sum of unique operators and unique operands in a method. It gives a sense of the complexity among the statements – for example a method which repeatedly uses a small number of variables is less complex than the method which uses a large number of different variables.
- *HVOL*, Halstead volume, is computed using *HLTH* and *HVOC* and can be seen as a measure of how much information does a reader need to absorb to understand a method. A small number of statements with a high Halstead Volume would suggest that the individual statements are quite complex.
- *HDIF*, Halstead Difficulty, uses a formula based on the number of unique operators, the number of unique operands, and the total number of operands to assess the complexity of a method. It suggests how difficult the code is to write and maintain.
- *HEFF*, Halstead effort, attempts to estimate the amount of work that it would take to recode a particular method.
- *HBUG*, Halstead bugs, attempts to estimate the number of bugs in a method.

For all these *Complexity* metrics, a higher value would mean higher code complexity, therefore low values are desirable.

Modularity Metrics. It is another category of metrics, that pertains to encapsulation and modularity of software.

- *CREF*, number of classes referenced, measures the total number external classes referenced by the method.
- *XMET* measures the total number of external methods called by a method. External methods are the methods that belong to a class other than the class in which the caller method is defined.
- *LMET* totals the number of local methods called by a method.
- *EXCR* and *EXCT* measure the total number of exceptions referenced and the total number of exceptions thrown by a method respectively.
- *MOD* is the total number of modifiers (public, private, abstract, static etc.) applied to the signature of a method.

For all of these *Modularity* metrics except for *MOD*, a higher value would mean that the code is less modular and therefore low values are desirable. We cannot find in literature what should be the desirable value for the *MOD* metric.

Documentation Metrics. These metrics capture properties of code-as-text.

- *NOCL* counts the total number of comment lines in a method.
- *NOC* measures the total number of comments in a method. It is different from *NOCL* as *NOC* counts a comment spread over, say 3 lines, as 1. Whereas *NOCL* will count it as 3.

A higher value for these metrics would mean that the code is documented more. A more documented code is easier to follow and therefore a higher value for these metric is more desirable. We note that we did not qualitatively looked into the comments and only conducted a quantitative analysis.

Most of the *complexity* and the *modularity* metrics (*COMP*, *HLTH*, *HVOC*, *HVOL*, *HDIF*, *HEFF*, *NOPR*, *NAND*, and *HBUG*) are the standard software quality metrics suggested by McCabe [54], and Halstead [55].

The metrics *CREF*, *XMET*, and *LMET* are derived from the object-oriented (OO) metrics that were suggested by Chidamber and Kemerer [56]. Basili et al. found that OO metrics appeared to be useful for predicting defect density [57]. Subramanyam and Krishnan conducted a survey on eight more empirical studies and showed that OO metrics are significantly associated with defects [58].

Additionally, some of these metrics (*NOS*, *MDN*, *TDN*, *CREF*, *XMET*, *LMET*, *HLTH*, *NOC*, *NOCL*, *CREF*, *XMET*, *LMET*, *NEXP*, *NLOC*) are also part of the SQO-OSS quality model.

Table I shows an additional piece of information for each metric. First, an up or down arrow near the metric name indicates what “better” means for that specific metric; this judgment comes either from studies that have been reported in the literature or from the intuitions that drove the definition of the metric in the first place. When those judgments cannot be made, there is no arrow.

IV. STUDY RESULTS

In this section, we present the results of our analyses of code clones and their relationship with each of the three software quality metric categories.

RQ1: In a project, are cloned methods more or less complex than non cloned methods?

We carried out two one-sided Wilcoxon Rank-Sum tests for each subject project and *Complexity* metrics. One Wilcoxon Rank-Sum test tested the hypothesis that cloned methods have lower (better) complexity than the non cloned method. The other test tested the hypothesis that non cloned methods have lower (better) complexity than the cloned methods. The summary of the two tests are shown in the Figure 4. The vertical axis shows the metrics. A number inside the parenthesis shows the number of projects for which we observed statistically significant test results ($p < 0.05$) and also at least a small effect size (Cohen’s d ; $r > 0.1$). The horizontal axis shows the percentage of these projects where we observed lower metric values for clones or non clones.

To understand the chart consider an example of *CAST* metric. *CAST(1570)* means that for the metric *Cast* there were 1570

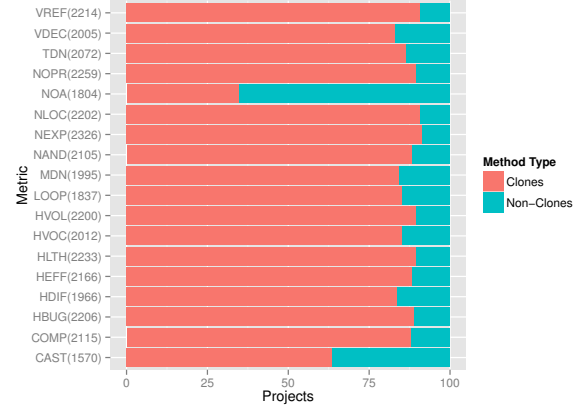


Fig. 4: Wilcoxon Rank-Sum test: stacked bar chart comparing cloned and non cloned methods. The Vertical-axis shows the *Complexity* metrics along with the number (in parenthesis) of subject projects that showed significant difference between cloned and non cloned methods ($p < 0.05$, $r > 0.1$).

projects significant difference ($p < 0.05$ and $r > 0.1$) in the cloned and non cloned methods. The horizontal red bar shows that in around 65% of these 1570 projects cloned methods have lower *CAST* metric values than the non cloned methods. The blue bar, on the other hand shows that in around 35% of these 1570 projects non cloned methods have lower *CAST* values.

To summarize, out of 4,221 projects, 1,804 to 2,121 projects showed significant difference ($p < 0.05$) in the complexity of cloned methods to that of the non cloned methods. For most of the metrics, except the *NOA* metric, more than 60% of the projects showed that cloned methods have lower complexity metric values than the non cloned methods.

221 projects, 673 to 2220 projects, showed significantly lesser ($p < 0.05$) values for *Complexity* metrics for cloned methods to that of the non cloned methods. The summary of the results for these tests are shown in Figure 4. A small percentage (4% to 10.5%) of projects showed negligible effect-size. Most of the projects showed small effect-size ranging from 54.7% to 64.4%. 24.2% to 30.8% of the projects showed medium effect and only 1% to 8.7% projects showed large effect. In total, 630 to 2,131 projects showed that complexity of cloned methods is significantly (P -value < 0.05 and $r > 0.1$) better than that of the non cloned methods, with at least small effect-size.

RQ2: In a project, are cloned methods more or less modular than non cloned methods?

We carried out two one-sided Wilcoxon Rank-Sum tests for each subject project and *Modularity* metrics. One Wilcoxon Rank-Sum test tested the hypothesis that cloned methods have lower (better) modularity metric values than the non cloned method. The other test tested the hypothesis that non cloned methods have lower (better) modularity than the cloned methods. The summary of the two tests are shown in the Figure 5; the axes description is similar to that of the Figure 4.

Out of 4,221 projects, 1,588 to 2,148 projects showed significant difference ($p < 0.05$, $r > 0.1$) in the modularity of

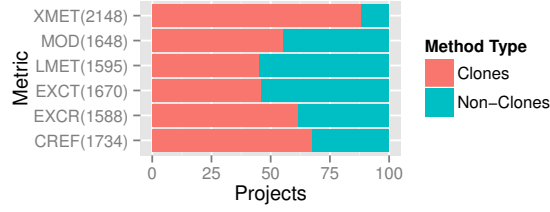


Fig. 5: Wilcoxon Rank-Sum test: stacked bar chart comparing cloned and non cloned methods. The Vertical-axis shows the *Modularity* metrics along with the number (in parenthesis) of subject projects that showed significant difference between cloned and non cloned methods ($p < 0.05$, $r > 0.1$).

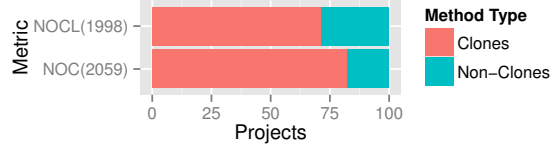


Fig. 6: Wilcoxon Rank-Sum test: stacked bar chart comparing cloned and non cloned methods. The Vertical-axis shows the *Documentation* metrics along with the number (in parenthesis) of subject projects that showed significant difference between cloned and non cloned methods ($p < 0.05$, $r > 0.1$).

cloned methods to that of the non cloned methods. In more than 50 % of the projects, the cloned methods performed better on *XMET*, *MOD*, *EXCR*, and *CREF* metrics. Whereas, the non cloned methods performed better on the *LMET* and *EXCT* metrics.

RQ3: In a project, are cloned methods more or less documented than non cloned methods?

Like the above two RQs, we conducted two one-sided Wilcoxon Rank-Sum tests for the two *Documentation* metrics. For *NOC*, 2,059 projects showed significant difference ($p < 0.05$, $r > 0.1$), whereas 1,998 projects showed significant difference in cloned and non cloned methods.

The summary of the results for these tests are shown in Figure 6. For both of these metrics, more than 70 % of the projects showed that the cloned methods have lesser code comments than the non cloned methods.

Form the above tests, for the projects where a significant difference was observed between clones and non clones, most of the projects showed that the cloned methods are more modular and less complex than the non cloned methods. Of the three quality metric categories, *Documentation* is the only one which showed negative results for cloned methods in comparison to the non cloned methods.

Controlling for size:

Many observable program features correlate strongly with code size. This knowledge has been used pervasively in quantitative studies of software through practices such as normalization on size metrics [59]–[62].

The absolute median difference in the number of statements of cloned and non cloned methods is found to be 47, which could negatively impact the study. To address this, we randomly selected an equal number of cloned and non cloned methods from each project. For example, consider a project

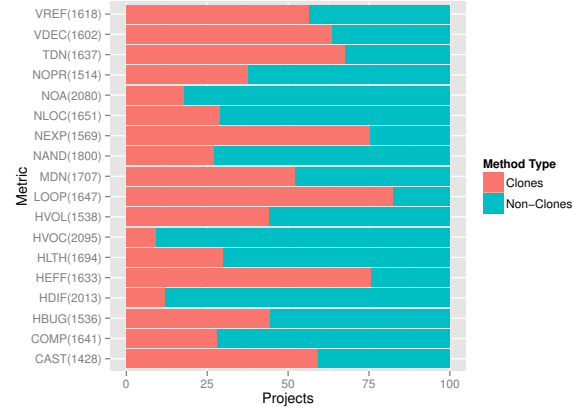


Fig. 7: Wilcoxon Rank-Sum test: stacked bar chart comparing cloned and non cloned methods. The Vertical-axis shows the normalized *Complexity* metrics along with the number (in parenthesis) of subject projects that showed significant difference between cloned and non cloned methods ($p < 0.05$, $r > 0.1$).

S, with *C1* number of cloned methods and *C2* number of non cloned methods, such that $C1 < C2$. In order to match the number of cloned methods with non cloned methods, we randomly selected *C1* out of *C2* non cloned methods. Similarly, consider a project *S2*, with *C1* number of cloned methods and *C2* number of non cloned methods, such that $C2 < C1$. In this case, we randomly selected *C2* number of cloned methods from the original pool of *C1* cloned methods.

The median *NLOC* of cloned methods and non cloned methods are 11 and 14 respectively. Also, the median *NOS* of cloned methods and non cloned methods are 8 and 10 respectively, suggesting that on an average, the cloned methods are 20% smaller than the non cloned methods. In order to make sure that the size is not impacting our study, we decided to redo our experiments using two size control strategies: i) dividing each metric by size; and ii) regression. We explain each of these strategies and the results below.

A. Dividing each metric by the size metric- *NOS*

In this strategy, for every method we divide each of the metrics with the method's *NOS* metric. For example, let us say for a method *m*, we have its *COMP* metric value to be 10 and its *NOS* metric value to be 50. Then the new *COMP* value of *m* becomes $10/50 = 0.2$.

Also, for each project, we matched the number of cloned methods to the non cloned methods. Two one-sided Wilcoxon Rank-Sum Tests were then carried out for each of the 3 categories of metrics. The summary of the results of these tests are shown in Figures 7 to 9.

These figures paint a different picture from the non-size-control experiments. Unlike the no size-control experiments, the cloned-methods do not appear to win, in-fact for most of the metrics in the *Complexity* and *Modularity* group of metrics, the blue bars (non clones) are longer than the red bars (clones), signifying that the non clones have better metric values than the cloned methods. In the documentation metrics group, the clones have performed worse on *NOC* metric. Whereas it appears to be a tie between clones and

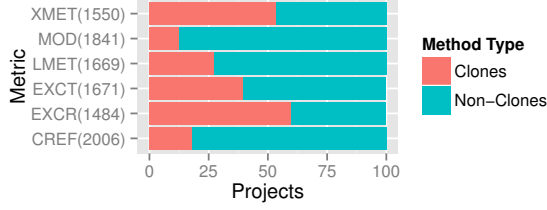


Fig. 8: Wilcoxon Rank-Sum test: stacked bar chart comparing cloned and non cloned methods. The Vertical-axis shows the normalized *Modularity* metrics along with the number (in parenthesis) of subject projects that showed significant difference between cloned and non cloned methods ($p < 0.05$, $r > 0.1$).

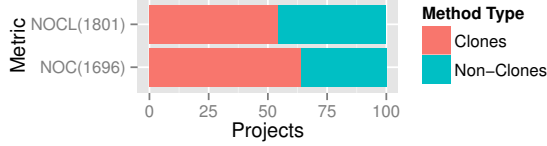


Fig. 9: Wilcoxon Rank-Sum test: stacked bar chart comparing cloned and non cloned methods. The Vertical-axis shows the normalized *Documentation* metrics along with the number (in parenthesis) of subject projects that showed significant difference between cloned and non cloned methods ($p < 0.05$, $r > 0.1$).

non clones for the *NOCL* metric.

A recent study on Java projects by Lopes and Ossher shows that diving a metric by size is not a best way to control for size [63]. They found that certain characteristics of programs vary disproportionately with program size, sometimes even non-monotonically. This could mean that dividing each metric by the size metric might not be making the metrics independent of size. To verify this, we plotted the scatter plots between the quality metrics and the size metric- *NOS*, both before and after normalization of the quality metric. We observed for many cases that the normalization of metrics does not remove the relationship between the metric and the size (*NOS*). The Figure 10 shows the relationship between 5 such randomly selected metrics and *NOS*, before and after normalization respectively. Only for *HBUG* and *MDN* metrics, we observe that the normalization has (almost) removed the relationship (see the fitted lines). In the case of other three metrics, normalization was not able to remove the relationship; in fact it has reversed the direction of the relationship. This exercise demonstrates that dividing each metric with size does not always make the metrics independent of size. Therefore, we need another method to control for size.

B. Regression

We use Linear Regression Models (LRM) with two independent variables to control for the confounding effect of size as explained by Kutner et. al. in their book [64]. The two independent variables used are *NOS*, a quantitative variable quantifying the size of the methods and *isClone*, a binary qualitative variable indicating if a method is a cloned method (*isClone* = 1) or not (*isClone* = 0). For each project, we built LRMs for each of the metric (excluding *NOS* metric) as

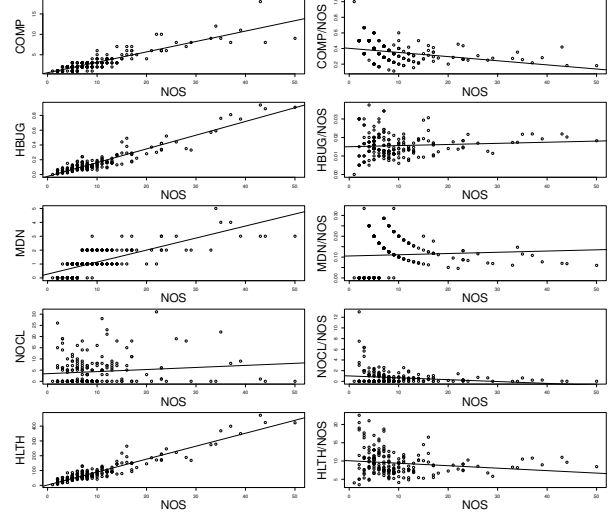


Fig. 10: Scatter plots showing relationship between quality metrics on vertical-axis and size metric (*NOS*) on horizontal-axis, before and after carrying out normalization of metrics. The plots in the left column show the relationship between the metric and the size before normalization, whereas the plots on the right show the relationship between the normalized metrics and the size metric.

a response variable, giving us a total of $4421 \times 26 = 114946$ LRMs. Out of these 114946 LRMs we selected the ones where R-squared values were greater than or equal to 65%. This step ensured that we are using only those LRMs which could fit the data well enough to explain at least 65% of the variance in the given response variable. We further rejected those LRMs where the presence of *isClone* variable was statistically insignificant ($p > 0.05$) in explaining the variance of the response variable. Moreover, we rejected the LRMs where the size of the effect (r), calculated using Hedge's g formula, was less than 0.1. In order to understand how LRMs are used to carry out size-control, consider the following example: In this example, the response variable is *XMET* metric, the two independent variables are *NOS* and *isClone* and the regression model is of the form:

$$E(XMET) = b_0 + b_1 \times NOS + b_2 \times isClone$$

where b_0 , b_1 , and b_2 are the parameters of the regression model. After fitting the regression model we get the following equation:

$$E(XMET) = 1.5510 + 0.42 \times NOS - 1.91 \times isClone$$

For the non cloned methods (*isClone*=0), the equation becomes:

$$E(XMET) = 1.55 + 0.42 \times NOS$$

whereas for the cloned methods (*isClone*=1), the equation becomes:

$$E(XMET) = -0.360251 + 0.42 \times NOS$$

Figure 11 shows the fitted regression lines for cloned and non cloned methods for the above example. Notice that both lines have the same slope ($b_1 = 0.42$), and therefore for any given level of *NOS*, the average distance between the two lines, given a fixed value of *NOS* is $b_2 = -1.91$ units (– sign indicating that cloned methods have lower *XMET* values than the non cloned methods). Thus, -1.91 is the average difference between the cloned methods and non cloned methods, when

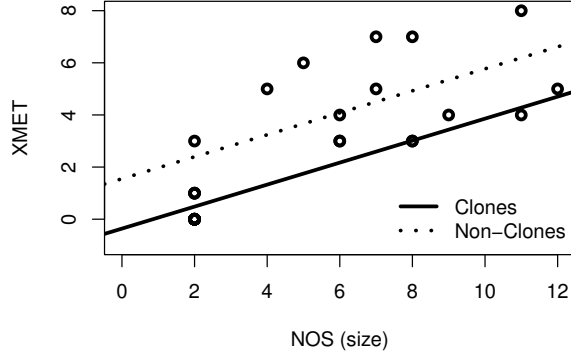


Fig. 11: Fitted regression lines for cloned and non cloned methods showing average difference between their XMET values. The vertical-axis shows the XMET metric values and the horizontal-axis shows the size of the methods as measured using NOS metric.

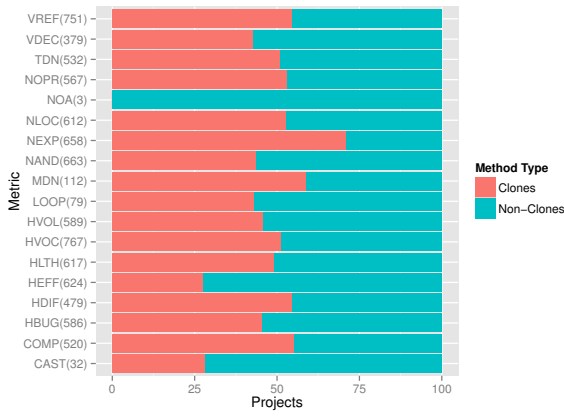


Fig. 12: Regression: stacked bar chart comparing cloned and non cloned methods. The Vertical-axis shows the Complexity metrics along with the number (in parenthesis) of subject projects that showed significant difference between cloned and non cloned methods ($p < 0.05$, $r > 0.1$).

controlling for size. The R-squared value of the above model is 84% ($> 65\%$), the P-value of the *isClone* variable is 0.002 (< 0.05) and the absolute effect size is 0.61 (> 0.1). This model satisfies all our criteria and therefore we can use this model in our analysis.

The Figures 12, 13, and 14 show the results of the size control experiments using regression. For the Complexity (Figure 12) and Documentation (Figure 14) categories of metrics it becomes clearly evident that the number of projects showing that cloned methods have lesser metric values is almost equal to the number of projects where non cloned methods have lesser metric values. The metrics NOA(3) and CAST(32) have a very few number of projects and therefore we can safely ignore them from this analysis. The metrics HEFF and NEXP are the only two metrics where we see a clear difference between cloned and non cloned methods. The cloned methods have won in NEXP metric, indicating that there are fewer number of expressions in the cloned methods, whereas the non cloned methods have won in the HEFF metric, indicating that the effort to code non cloned methods is lower than the effort required to code the cloned methods.

In the Modularity (Figure 13) category of metrics, it ap-

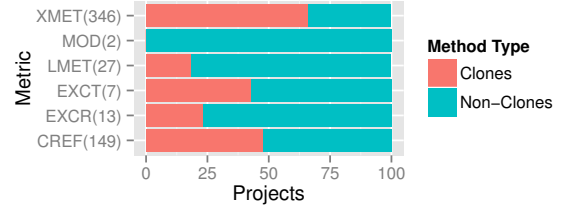


Fig. 13: Regression: stacked bar chart comparing cloned and non cloned methods. The Vertical-axis shows the Modularity metrics along with the number (in parenthesis) of subject projects that showed significant difference between cloned and non cloned methods ($p < 0.05$, $r > 0.1$).

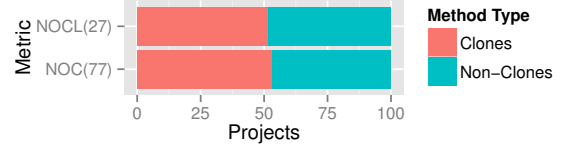


Fig. 14: Regression: stacked bar chart comparing cloned and non cloned methods. The Vertical-axis shows the Documentation metrics along with the number (in parenthesis) of subject projects that showed significant difference between cloned and non cloned methods ($p < 0.05$, $r > 0.1$).

pears that the non cloned methods have lesser metric values (MOD(2), LMET(27), EXCR(13), and EXCT(7)) and therefore non cloned methods appear to be more modular than the cloned methods. However, a very small number of projects (shown in the parenthesis) have shown these effects. For the XMET metric, around 70% of the 346 projects have shown that cloned methods make lesser calls to the external methods than the non cloned methods, indicating that the cloned methods in these projects are more modular.

The results from this experiment are very different from the non-size-control experiment. We summarize the regression experiment results. *i) Only upto 17% of the projects from the population of 4,421 projects show statistically significant difference between the cloned and the non cloned methods with at least a small effect for any of the quality metrics; and ii) The number of projects for which cloned methods have statistically significant ($p < 0.05$, $r > 0.1$) lower metric values is almost equal to the number of projects for which non cloned methods have statistically significant lower metric values.*

V. LIMITATIONS OF THIS STUDY

The results of this study are from a large number of open source software projects written in Java. Although we chose subject projects that exhibit variety in their size and type, several factors such as development practices within OSS communities, release cadence, and team size might have impact on the observed phenomena. Thus drawing general conclusions from empirical studies such as this is difficult because the results depend on many relevant confounding variables [65]. For this reason, we cannot assume a priori that the results of this study generalize beyond the setting for which it was conducted. However, the overall results of this study showed several commonalities across a wide range of OSS Java projects and indicate that the results hold for more than just the studied projects.

VI. CONCLUSION AND FUTURE WORK

We conducted an statistical study to explore if there exists any difference between the quality of cloned methods and non cloned methods, as measured by well-known software quality metrics. The dataset consists of 4,421 open source Java projects containing 644,830 cloned and 842,052 non cloned methods. The study uses 27 software metrics, as a proxy for quality, spanning across complexity, modularity, and documentation categories.

Statistically we find no evidence that the cloned methods are different from the non cloned methods for most of the metrics. The main exception pertained to size: we found that cloned methods, on an average (median *NOS*), are 20% smaller than the non cloned methods, which may indicate that developers are careful to copy-paste only small(er) pieces of code. A small fraction of the projects show statistically significant difference in the quality of cloned and non cloned methods, when controlled for size. There, however, are only three metrics *NEXP*, *HEFF*, and *XMET* where the difference between the percentage of projects showing that the cloned methods have better metrics values and the percentage of projects showing that the non cloned methods have better metrics is at least 30%. Also, our observations resonate with other studies in finding that size is a confounding variable in such studies. Similar to what is described in [63], normalizing the metrics for size didn't always make the metrics completely independent of size.

There are a number of directions for future work suggested by the results and limitations of our study. First, the set of software quality metrics could be further expanded to capture additional characteristics of cloned and non cloned methods.

Including other source code repositories into the study is another area for future work. While the Maven repository represents a large fraction of the open source Java community, additional open source repositories like SourceForge, Google Code, and Apache would add greater diversity to the dataset. Especially of interest are those repositories based on more recent version control projects like Git and Mercurial, which have seen a surge in popularity in recent years. GitHub and Bitbucket are prime candidates, as they now hosts many popular open source projects. Another longer term goal is to expand the dataset with projects written in programming languages in addition to Java.

The data collected for our study can also be used in numerous ways, in addition to further analysis of the primary question. For example, our extremely large collection of computed software quality metrics and the output of clone detector, which could be used to study the evolution of metrics in both cloned and non cloned methods. Additionally, the software quality data could be produced at different granularities such as class level, package level, and project level, which could be associated with open source community data to investigate if any metrics are related to properties of the surrounding open source community. Moreover, the software quality data at different granularities and at various versions of a project

could be used to study the project evolution from different view points.

Dataset accessibility and Reproducibility: We welcome other researchers to reproduce and even replicate the study on other software projects. For this, we have made available all the necessary artifacts including raw data, detailed steps and scripts to process data, and analysis procedure to reproduce the statistical results to verify the claims. All artifacts are made available at the following URL: <http://mondego.ics.uci.edu/projects/clone-metrics>.

REFERENCES

- [1] C. K. Roy, J. R. Cordy, and R. Koschke, "Comparison and evaluation of code clone detection techniques and tools: A qualitative approach," *Sci. Comput. Program*, pp. 470–495, 2009.
- [2] D. L. Parnas, "On the criteria to be used in decomposing systems into modules," *Commun. ACM*, vol. 15, no. 12, pp. 1053–1058, Dec. 1972.
- [3] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts, *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 1999.
- [4] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier, "Clone Detection Using Abstract Syntax Trees," in *Proceedings of the International Conference on Software Maintenance*. IEEE Computer Society, 1998, p. 368.
- [5] B. Baker, "A program for identifying duplicated code," *Computing Science and Statistics*, pp. 24–49, 1992.
- [6] S. Ducasse, M. Rieger, and S. Demeyer, "A Language Independent Approach for Detecting Duplicated Code," in *Proceedings of the IEEE International Conference on Software Maintenance*. IEEE Computer Society, 1999, p. 109.
- [7] J. H. Johnson, "Identifying redundancy in source code using fingerprints," in *Proceedings of the 1993 conference of the Centre for Advanced Studies on Collaborative research: software engineering - Volume 1*. Toronto, Ontario, Canada: IBM Press, 1993, pp. 171–183.
- [8] T. Kamiya, S. Kusumoto, and K. Inoue, "CCFinder: a multilingual token-based code clone detection system for large scale source code," *IEEE Trans. Softw. Eng.*, vol. 28, no. 7, pp. 654–670, 2002.
- [9] R. Komondoor and S. Horwitz, "Using Slicing to Identify Duplication in Source Code," in *Proceedings of the 8th International Symposium on Static Analysis*. Springer-Verlag, 2001, pp. 40–56.
- [10] J. Krinke, "Identifying Similar Code with Program Dependence Graphs," in *Proceedings of the Eighth Working Conference on Reverse Engineering (WC'RE'01)*. IEEE Computer Society, 2001, p. 301.
- [11] N. Gode and R. Koschke, "Incremental clone detection," in *Proceedings of CSMR*, 2009.
- [12] R. Koschke, "Large-scale inter-system clone detection using suffix trees," in *Proceedings of CSMR*, 2012, pp. 309–318.
- [13] R. Komondoor and S. Horwitz, "Effective automatic procedure extraction," in *Proceedings of the International Workshop on Program Comprehension*. Springer-Verlag, 2003, pp. 40–56.
- [14] R. Koschke, "Identifying and removing software clones," pp. 15–36, 2008.
- [15] M. Shomrat and Y. Feldman, "Detecting refactored clones," in *ECOOP 2013 European Conference on Object-Oriented Programming*, ser. Lecture Notes in Computer Science, G. Castagna, Ed. Springer Berlin Heidelberg, 2013, vol. 7920, pp. 502–526.
- [16] C. Kasper and M. Godfrey, "'cloning considered harmful' considered harmful: patterns of cloning in software," *Empirical Software Engineering*, vol. 13, no. 6, pp. 645–692, 2008.
- [17] M. Kim, V. Sazawal, D. Notkin, and G. Murphy, "An empirical study of code clone genealogies," in *Proceedings of FSE*, 2005.
- [18] C. K. Roy and J. R. Cordy, "A survey on software clone detection research," *Technical Report*, Queen's University at Kingston, 2007.
- [19] R. Koschke, "Survey of research on software clones," in *Proceedings of Duplication, Redundancy, and Similarity in Software*, 2007.
- [20] F. Rahman, C. Bird, and P. Devanbu, "Clones: what is that smell?" *Empirical Software Engineering*, vol. 17, no. 4-5, pp. 503–530, 2012.

- [21] H. Sajnani, V. Saini, and C. V. Lopes, "A comparative study of bug patterns in Java cloned and non-cloned code," in *Source Code Analysis and Manipulation (SCAM), 2014 IEEE 14th International Working Conference on*. IEEE, 2014, pp. 21–30.
- [22] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, and E. Merlo, "Comparison and evaluation of clone detection tools," *Software Engineering, IEEE Transactions on*, vol. 33, no. 9, pp. 577–591, Sept 2007.
- [23] J. H. Johnson, "Substring matching for clone detection and change tracking," in *International Conference on Software Maintenance*, 1994, pp. 120–126.
- [24] A. Lozano, M. Wermelinger, and B. Nuseibeh, "Evaluating the harmfulness of cloning: a change based experiment," in *Mining Software Repositories*, 2007, pp. 18–22.
- [25] J. Krinke, "A study of consistent and inconsistent changes to code clones," in *Working Conference on Reverse Engineering*, 2007, pp. 170–178.
- [26] S. Xie, F. Khomh, Y. Zou, and I. Keivanloo, "An empirical study on the fault-proneness of clone migration in clone genealogies," in *Proc. of CSMR-WCRE*. IEEE, 2014, pp. 94–103.
- [27] E. Juergens, F. Deissenboeck, B. Hummel, and S. Wagner, "Do code clones matter?" in *Proceedings of ICSE*, 2009, pp. 485–495.
- [28] J. Ossher, H. Sajnani, and C. V. Lopes, "File cloning in open source Java projects: The good, the bad, and the ugly," in *ICSM*. IEEE, 2011.
- [29] M. de Wit, A. Zaidman, and A. van Deursen, "Managing code clones using dynamic change tracking and resolution," in *Proceedings of the 25th International Conference on Software Maintenance (ICSM 2009)*. IEEE Computer Society, 2009.
- [30] J. Cordy, "Comprehending reality - practical barriers to industrial adoption of software maintenance automation," in *Proceedings of International Conference on Program Comprehension*, 2003, pp. 196–205.
- [31] Rajapakse, D. C., and S. Jarzabek, "Using server pages to unify clones in web applications: A trade-off analysis," in *Proceedings of International Conference on Software Engineering*, 2007, pp. 116–126.
- [32] M. Bruntink, A. van Deursen, R. van Engelen, and T. Tourwé, "On the use of clone detection for identifying cross cutting concern code," *IEEE Transactions on Software Engineering*, vol. 31, no. 10, pp. 804–818, 2005.
- [33] V. R. Basili, L. C. Briand, and W. L. Melo, "A Validation of Object-Oriented Design Metrics as Quality Indicators," *IEEE Trans. Softw. Eng.*, vol. 22, no. 10, pp. 751–761, 1996.
- [34] D. Kafura and G. Reddy, "The use of software complexity metrics in software maintenance," *Software Engineering, IEEE Transactions on*, vol. SE-13, no. 3, pp. 335–343, March 1987.
- [35] W. Li and S. Henry, "Object-oriented metrics that predict maintainability," *J. Syst. Softw.*, vol. 23, no. 2, pp. 111–122, Nov. 1993.
- [36] A. Maven, <http://maven.apache.org/>.
- [37] C. Lopes, S. Bajracharya, J. Ossher, and P. Baldi, "UCI source code data sets," 2010. [Online]. Available: [http://www.ics.uci.edu/~sim\\$lopes/datasets/](http://www.ics.uci.edu/~sim$lopes/datasets/)
- [38] H. Sajnani, V. Saini, J. Svajlenko, C. K. Roy, and C. V. Lopes, "Sourcecrcc: Scaling code clone detection to big code," in *Proceedings of International Conference on Software Engineering*, 2016, to appear.
- [39] "Jhawk," <http://mondego.ics.uci.edu/projects/clonedetection/>.
- [40] V. Gupta, K. Aggarwal, and Y. Singh, "A fuzzy approach for integrated measure of object-oriented software testability," *Journal of Computer Science*, vol. 1, no. 2, pp. 276–282, 2005.
- [41] J. S. Alghamdi, R. A. Rufai, and S. M. Khan, "Oometer: A software quality assurance tool," in *null*. IEEE, 2005, pp. 190–191.
- [42] H. C. Benestad, B. Anda, and E. Arisholm, "Assessing software product maintainability based on class-level structural measures," in *Product-Focused Software Process Improvement*. Springer, 2006, pp. 94–111.
- [43] E. Arisholm and L. C. Briand, "Predicting fault-prone components in a Java legacy system," in *Proceedings of the 2006 ACM/IEEE international symposium on Empirical software engineering*. ACM, 2006, pp. 8–17.
- [44] E. Arisholm, L. C. Briand, and M. Fuglerud, "Data mining techniques for building fault-proneness models in telecom Java software," in *Software Reliability, 2007. ISSRE'07. The 18th IEEE International Symposium on*. IEEE, 2007, pp. 215–224.
- [45] M. Andersson and P. Vestergren, "Object-oriented design quality metrics," 2004.
- [46] E. Nasser, S. Counsell, and M. Shepperd, "An empirical study of evolution of inheritance in Java OSS," in *Software Engineering, 2008. ASWEC 2008. 19th Australian Conference on*. IEEE, 2008, pp. 269–278.
- [47] A. Mubarak, S. Counsell, and R. Hierons, "Does an 80: 20 rule apply to Java coupling?" in *Proceedings of the International Conference on Evaluation and Assessment in Software Engineering*, 2009.
- [48] A. Mubarak, S. Counsell, and R. M. Hierons, "An evolutionary study of fan-in and fan-out metrics in OSS," in *Research Challenges in Information Science (RCIS), 2010 Fourth International Conference on*. IEEE, 2010, pp. 473–482.
- [49] J. Börstler, M. Nordström, and J. H. Paterson, "On the quality of examples in introductory Java textbooks," *ACM Transactions on Computing Education (TOCE)*, vol. 11, no. 1, p. 3, 2011.
- [50] D. Lucrédio, E. S. de Almeida, and R. P. Fortes, "An investigation on the impact of mde on software reuse," in *Software Components Architectures and Reuse (SBCARS), 2012 Sixth Brazilian Symposium on*. IEEE, 2012, pp. 101–110.
- [51] R. Scandariato and J. Walden, "Predicting vulnerable classes in an android application," in *Proceedings of the 4th international workshop on Security measurements and metrics*. ACM, 2012, pp. 11–16.
- [52] K. Herzig, S. Just, A. Rau, and A. Zeller, "Classifying code changes and predicting defects using change genealogies," Technical Report, Tech. Rep.
- [53] N. Nagappan, T. Ball, and A. Zeller, "Mining metrics to predict component failures," in *Proceedings of the 28th international conference on Software engineering*. ACM, 2006.
- [54] T. J. McCabe, "A complexity measure," *Software Engineering, IEEE Transactions on*, no. 4, pp. 308–320, 1976.
- [55] M. H. Halstead, *Elements of Software Science (Operating and programming systems series)*. Elsevier Science Inc., 1977.
- [56] S. R. Chidamber and C. F. Kemerer, "A Metrics Suite for Object Oriented Design," *IEEE Trans. Softw. Eng.*, vol. 20, no. 6, pp. 476–493, 1994.
- [57] V. R. Basili, G. Caldiera, and H. D. Rombach, "The goal question metric approach," *Encyclopedia of software engineering*, vol. 1, pp. 528–532, 1994.
- [58] R. Subramanyam and M. S. Krishnan, "Empirical analysis of ck metrics for object-oriented design complexity: Implications for software defects," *IEEE Trans. Softw. Eng.*, vol. 29, no. 4, pp. 297–310, Apr. 2003. [Online]. Available: <http://dx.doi.org/10.1109/TSE.2003.1191795>
- [59] K. El Emam, S. Benlarbi, N. Goel, and S. N. Rai, "The confounding effect of class size on the validity of object-oriented metrics," *Software Engineering, IEEE Transactions on*, vol. 27, no. 7, pp. 630–650, 2001.
- [60] M. Cartwright and M. Shepperd, "An empirical investigation of an object-oriented software system," *Software Engineering, IEEE Transactions on*, vol. 26, no. 8, pp. 786–796, 2000.
- [61] L. C. Briand, J. Wüst, J. W. Daly, and D. V. Porter, "Exploring the relationships between design measures and software quality in object-oriented systems," *Journal of systems and software*, vol. 51, no. 3, pp. 245–273, 2000.
- [62] H. Sajnani, V. Saini, J. Ossher, and C. Lopes, "Is popularity a measure of its quality? an analysis of maven components," in *Proceedings of the 30th Software Maintenance and Evolution (To appear in ICSME 2014)*. IEEE Computer Society, 2014.
- [63] C. V. Lopes and J. Ossher, "How scale affects structure in Java programs," in *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*. ACM, 2015, pp. 675–694.
- [64] K. M., N. C., N. J., and L. W., *Applied Linear Statistical Models*. Sage Publications.
- [65] V. R. Basili, F. Shull, and F. Lanubile, "Building knowledge through families of experiments," *Software Engineering, IEEE Transactions on* vol. 25, no. 4, pp. 456–473, 1999.