

A Comparative Study of Bug Patterns in Java Cloned and Non-cloned Code

Hitesh Sajnani, Vaibhav Saini, and Cristina V. Lopes
Bren School of Information and Computer Sciences
University of California, Irvine, Irvine, CA, USA
Email: {hsajnani, vpsaini, lopes}@ics.uci.edu

Abstract—Code cloning via copy-and-paste is a common practice in software engineering. Traditionally, this practice has been considered harmful, and a symptom that some important design abstraction is being ignored. As such, many previous studies suggest approaches to facilitate the discovery, removal, and refactoring of clones. However, not many studies exist that empirically investigate the relationship of code clones with code quality.

In this paper, we conduct an empirical study of 31 open source Java projects (1.7 MSLOC) to explore the relationship between code clones and a set of bug patterns reported by FindBugs. We found that: (i) the defect density in cloned code is 3.7 times less than that of the rest of the code; (ii) 66% of the bug patterns associated with code clones are related to issues in coding style and practice, the two least problematic of the FindBugs categories, while that number is 49% for non-cloned code; and (iii) 75% of the bug patterns in cloned code are duplicated without any changes, while 25% are only present in one of the clones.

These results show that, when using FindBugs to detect bug patterns, there is a positive differentiation of cloned code with respect to the rest of the code: the cloned code has considerably less, and less problematic, bug patterns. While our study does not unveil any explanation for this, results from other, more qualitative studies indicate that developers use copy-and-paste intentionally and wisely, which may explain the quantitative observations of our study. Overall, these research results suggest that the practice of code cloning in Java, and possibly in all other object-oriented languages, needs to be given serious consideration on the part of tool designers.

I. INTRODUCTION

Over the last couple of decades, software development practices have changed drastically. Pervasive high-speed internet, full fledged IDEs, and a whole new generation of hyper-connected young programmers weaned on the web have established new programming practices based on massive collaboration. These days, it is easier than ever to find and use a well-tested piece of code written by someone else that does exactly what we want. Glueing these pieces together and putting them in the right context is still a necessary and an important skill. But, for better or for worse, copy-and-paste is no longer a pejorative term, but a factual observation about how part of modern coding gets done today. Reusing code fragments via copy-and-paste, with or without modifications or adaptations, also known as code cloning, has become a common behavior of software engineers [50].

Although pervasive, code cloning has traditionally been criticized by researchers and leading practitioners alike. Parnas [45] said that “if you use copy and paste while you’re coding, you’re probably committing a design error.” Indeed, if instead of copying code, we move it into its own method, future modifications will be easier because we will need to modify the code in only one location. The code will be more reliable because we will have only one place to ensure that the code is correct. Consequently, code cloning is often presented as a negative design characteristic in software systems. Considered as a bad “code smell” [16], a considerable amount of research in cloning is concentrated on detecting clones in existing source code [5], [2], [14], [23], [27], [30], [36], [17], [34], removing them [31], [33] and refactoring them [53].

Under many circumstances, code cloning can, indeed, be harmful. But it also has advantages like rapid development, reuse of tested code, and separation of concerns. The pervasive practice of code cloning has more recently attracted researchers to conduct empirical studies to find evidence about the effects (good or bad) of code cloning. Such attempts have questioned our conventional wisdom about the harmful nature of clones. For example, Kasper and Godfrey [28] presented evidence that clones may be intentional and that they improve developer productivity. Kim et al. [29] found that most of the clones are short lived – i.e. starter code that quickly becomes something else – and hence investment made in refactoring them may not be worth the effort. Toomim et al. [55] showed that managing clones via linked editing to edit multiple cloned regions without much programmer intervention can be an efficient way of dealing with clones. Rahman et al. [46] conducted a study on four subject systems to assess the impact of clones on defect occurrence of software products and did not find any evidence that cloning is harmful.

These findings present a different perspective on code cloning that has implications for the future research in this area. Although important, most of the research in this direction, so far, is either qualitative or performed on very few subject systems. An excellent survey [49] on code clone research mentions that “there is little information available concerning the impacts of code clones on software quality”, expressing the need to conduct more empirical studies examining the impact of code cloning on various factors related to code and other artifacts. Koschke [32] lists several important open issues in this field, one of which being “What is the relation of clones to quality attributes?”

To that end, we conduct an empirical study to explore the relation between code clones and bugs on 31 Java projects, totalling 1.7 MSLOC. Our study examines the relationship between clones and various bug patterns reported by FindBugs [15]. A bug pattern is a code idiom that is often a programming error and hence an indicator of code quality. These bug patterns arise for a variety of reasons including difficult language features, misunderstood API methods, misunderstood invariants when code is modified during maintenance, and variety of mistakes including typos, use of the wrong boolean operator, etc. Hence they capture various kinds of issues impacting the quality of code. Moreover, these bug patterns are organized into high level categories like *Bad Practice*, *Correctness*, *Performance*, etc. We use this categorization to establish the relationship between clones and specific bug categories. We posit that such analysis will help us to investigate the associations between code quality and cloning, which in turn, will be useful to inform research in this field.

More specifically, we seek answers to the following research questions.

Research Question 1: *Is defect density of cloned code greater than that of non-cloned code?*

In this study, we consider a piece of code to be cloned if there exists a similar piece of code in the same system (intra-system clones only).

We use bug patterns reported by FindBugs to calculate the defect density, where defect density is defined as the number

of bug patterns reported by FindBugs per 1,000 lines of code. FindBugs pro-actively reports likely defect locations in code by using range of approaches from simple code pattern-matching techniques to rigorous static analyses that process carefully designed semantic abstractions of code. Thus FindBugs *modus operandi* is to automatically prove certain properties of a program, i.e. certifying the program free of a certain class of bugs. Since, large scale studies like ours are heuristic in nature, FindBugs gives us the automation we need for this much larger dataset. We discuss more about the rationale of using FindBugs' bug patterns in the context of this study in II-C.

This research question is the main topic of our study. Clearly, if we were to find a much higher defect density in cloned code than in the rest of the code, there would be irrefutably strong arguments against the practice of cloning.

Research Question 2: *Are there specific bug categories which are seen more often in the cloned code?*

Each bug pattern reported by FindBugs is associated with a category. Since each category poses a different threat level and captures a different class of bugs, we try to examine the relationship between these categories and code clones. Such analysis is useful for risk assessment and employing targeted measures to mitigate the risks.

If we were to find that cloned code has higher rate of *Correctness*, *Performance* or *Security* bug patterns than the rest of the code, that would also be a strong argument against the practice of cloning.

Research Question 3: *How often do bug-patterns propagate through cloning?*

The primary goal of this research question is to explore if code cloning leads to an increase in the total number of bugs in the system. If this is the case, each code block with a bug, when cloned, adds one more bug in the code-base; eventually degrading the overall code quality. One implication of this finding could be building tools and techniques to help developers pro-actively improve code snippets (e.g., fix bugs) before copying it to some other location.

Findings. We found that the defect density of code clones is in fact less than that of the code which is not cloned. We also found that bug patterns related to *Style*, and *Bad Practice* categories are more prevalent in code clones. These categories do not pose any severe threat to the code-base and only affect the readability of the code. Moreover, we found that about 75% of the times, a bug pattern is also duplicated when the code is cloned. But again, since these bug patterns are mostly associated with *Style*, and *Bad Practice* categories, it may not be difficult to add tool support to clean the code before cloning.

While our study is not the final word on the issue, it is one more piece of evidence that in the practice of software development, clones do not seem to be as bad as they have been thought to be. 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. But if clones are not as bad as we thought they were, this leads to interesting new avenues of exploration for tools that help *manage* clones rather than *eliminating* them. Such tools and techniques can help developers take advantage of rapid development using cloning and manage clones automatically to avoid degrading the quality of cloned code.

Outline: The remainder of the paper describes the study design (Section II), results and examination of statistical differences (Section III), the threats to validity (Section IV), and related work that has been done in this area (Section V). Finally, we summarize our findings (Section VI).

II. STUDY DESIGN

At a high level, the overall approach to analyze the bug patterns in the code clones is a two step process. In the first

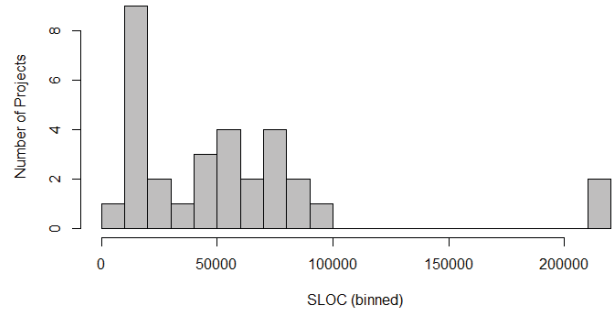


Fig. 1: Size distribution of projects. Size is defined as non-commented lines of code

step, we use a clone detection tool to identify all the clones present in each subject system. We set the granularity of the tool to detect method level clones. Thus, we have a list of methods for which clones exist in the system. Similarly, we also have a list of methods for which no clones exist in the system. A set of methods which are clones of each other is called a code clone group. Each member of a group is a clone sibling.

In the second step, we find all the bug patterns present in the subject system using FindBugs. After detecting all the bug patterns in the system, we create a map to associate methods with the bug patterns found in them. Since, each bug pattern belongs to a high level bug category, we also create a map to associate methods with bug categories.

Combining the result of the previous two steps, we have, for each method in the subject system:

- 1) All the other methods which are clones of this method (clone siblings); and
- 2) All the bug patterns and bug categories found in this method

We use this information and statistical analysis to seek answers to the research questions posed above. In the remainder of this section, we describe each aspect of our study design in detail.

A. Subject Systems

We chose 31 open source Apache Java projects as subject systems for this study. These projects are of varied size and span across various domains including search and database systems, server systems, distributed systems, machine learning and natural language processing libraries, and network systems. Most of these subject systems are highly popular in their respective domain. Such subjects systems exhibiting variety in size and domain help counter a potential bias of our study towards any specific kind of software system.

Figure 1 describes the size distribution of the subject systems. The X-axis represents the binned uncommented lines of code (SLOC). The Y-axis represents the number of projects. The average project size is 54,382 SLOC. The name and exact size of each project is listed in Column 1 & 2 of Table I respectively.

B. Clone Detection

Several techniques have been proposed for clone detection over many years [40], [21], [43], [27], [3], [5], [35], [22], [30], [36], [25], [34], [42], [11], [56]. These techniques differ in many ways ranging from the type of detection algorithm they use to the source code representation they operate on.

Techniques using various representations include Tokens [27], [3], Abstract Syntax Trees [5], [35], [22], Program

Dependence Graphs [30], [36], Suffix Tree [5], [25], [27], [34], Text representations [42], [11], Hash representations [56], etc. Each of these different approaches have their own merits and are useful for different use-cases. For example, AST based techniques have high precision, and are useful for refactoring or clone source manipulations, but they may not scale well. On the other hand, token based techniques have high recall but may yield clones which are not syntactically complete and hence impacting precision [6]. But with the choice of the right similarity function, and similarity threshold value, they have demonstrated high precision. Another advantage of token-based techniques is that they are highly scalable and fast.

We use our own scalable token-based clone detection tool presented in [51] which is capable of detecting Type 3 method level clones.¹ We have made the tool publicly available at [9].

We chose *overlap similarity* as the similarity function and similarity threshold value θ to be 0.8 because in our previous experiments these configurations gave maximum precision without impacting recall.

False Positives. In order to evaluate if the clones detected by the tool are legitimate, we chose a random sample of clone siblings for manual inspection. We examined 140 such clone siblings and classified them as either legitimate clones or false positives. We discovered only 4 false positives. Hence, using a t-test, we can conclude with 95% confidence that the false positive rate of the clone detection tool is only $2.3 \pm 5\%$.

On further examining the false positives, we found that the clone siblings were methods which are *stubs* without any functionality or methods of very small size. To further safely avoid such cases, we included only those methods which had at least 20 tokens in the body. This step only eliminated all the false positives in the previous sample further increasing precision without impacting recall.

Column 5 & 6 in Table I respectively list the *total methods* and *cloned methods* found in the subject systems. Whereas Column 4 & 7 show the percentage of total SLOC and percentage of total methods clones in each subject system.

C. Why use Bug Patterns?

Measuring external quality can be exceedingly challenging, as it requires identifying defects and inaccuracies in software. Such defect identification is usually done by either using bug reports or by using static analysis tools like FindBugs.

While both these techniques of defect identification have their merits [47], in the context of this study, there are several reasons why we choose bug patterns detected by FindBugs to measure the external code quality.

First, defect identification relies heavily on links between bug databases and program code repositories. This linkage is typically based on bug-fixes identified in developer-entered commit logs. Unfortunately, developers do not always report which commits perform bug-fixes. Moreover, developers sometimes fix bugs that are only reported in some other projects' bug tracker, rather than in their own; and vice-versa. Thus only a fraction of bug fixes are actually labelled in source code version histories which become available for the study. The question naturally arises, are the bug fixes recorded in these historical datasets a fair representation of the full population of bug fixes? Bird et al. [7] investigated historical data from several software projects, and found strong evidence of systematic bias. They found that bias is a critical problem that threatens both the effectiveness of processes that rely on biased datasets to build prediction models and the generalizability of hypotheses tested on biased data.

Second, apart from completeness of data, there are also serious concerns regarding the quality of bug-fix data obtained

in such manner. Herzig et al. [19] found 33.8% of all issue reports to be misclassified, that is, rather than referring to a bug fix, they resulted in a new feature, or an internal refactoring. This misclassification introduces bias in bug data, confusing bugs and features. They manually examined 7,000 issues, and found that on average, 39% of files marked as defective, actually never had a bug.

Third, and more importantly, the number of reported defects is heavily correlated to the popularity of a project [18]. The more users a project has, the more people there are discovering and reporting defects. A project, only used by a handful of people, may be significantly lower in quality than a project used by tens of thousands of people, but popular projects will almost certainly have more bugs reported. However, in our recent study, we did not find any correlation between bug patterns and popularity [52].

Fourth, similar to the third issue, affects the number of reported bugs based on the life of the project. A very old project may have hundreds of reported bugs, however, a project that has just started may have fewer bugs, if any. Thus practically making it impossible to compare them.

FindBugs use heuristics and static analysis to identify common bug patterns in software that may result in externally visible defects. Any instances of these patterns are then reported to developers as potential bugs. While such systems make no guarantees as to correctness or completeness, studies have shown that they regularly identify important defects in software [1], [20]. Moreover, a recent study shows that using historical data for bug prediction using bug databases does not fare as well against FindBugs, generally doing worse [47]. Also, since large scale studies like ours are heuristic in nature, FindBugs gives us the automation we need for this much larger dataset. As a result, we use a proxy in the form of FindBugs [15], a popular static analysis tool in both research and industry. We believe that looking at the exercise of code cloning in this light will provide us with one more perspective to understand the practice of code cloning better.

Credibility of FindBugs. FindBugs is an heuristic tool; as such, it suffers from both false positives and false negatives. False positives (i.e. reported bugs that aren't really bugs) are particularly problematic. A previous study has shown that FindBugs results in slightly less than 50% false positives [20], which is a high rate. Nevertheless, the rate of false positives in FindBugs affects the cloned code and the non-cloned code in about the same way. Since our goal is a comparison of these two sets of code, our use of FindBugs is fair.

Second, FindBugs bug patterns are used to assess open source software on a regular basis. For example, in order to regularly perform scans of open source software, the U.S. Department of Homeland Security uses Coverity [12], a commercial high-end bug finding product, which includes several bug patterns from FindBugs.

Third, in 2009, Google held a global fixit for FindBugs tool that had interesting results². The focus of the fixit was to get feedback on the 4,000 highest confidence issues found by FindBugs at Google, and let Google engineers decide which issues, if any, needed fixing. More than 700 engineers ran FindBugs from dozens of offices. More than 250 of them entered more than 8,000 reviews of the issues. A review is a classification of an issue as must-fix, should-fix, mostly-harmless, not-a-bug, and several other categories. More than 75% of the reviews classified issues as must fix, should fix or I will fix. Many of the scariest issues received more than 10 reviews each. It is reported that engineers have already submitted changes that made more than 1,100 of the 3,800 issues go away. Engineers filed more than 1,700 bug reports, of which 600 have already been marked as fixed. Work continues on addressing the issues raised by the fixit, and on supporting

¹A Type 3 clone is a copy with further modifications in which statements have been changed, added, or removed [32].

²Information published on FindBugs website: <http://FindBugs.sourceforge.net/>

the integration of FindBugs into the software development process at Google.

These observations give confidence that the bug patterns reported by FindBugs, even if not measuring external quality directly and deterministically, are positively correlated with it.

Bug Pattern Categories. FindBugs classifies each bug pattern into a specific category which determines the type of the bug pattern. These categories include *Correctness*, *Multi-threaded Correctness*, *Performance*, *Security*, *Malicious Code*, *Style*, and *Bad Practice*. A short description of each bug pattern along with its category can be found at <http://FindBugs.sourceforge.net/bugDescriptions.html>.

Limiting False Positives. One of the problems with the code quality tools like FindBugs is that they tend to overwhelm developers with problems that may not really be problems i.e., false positives. Although FindBugs is reported to have less than 50% false positives [20], we believe that large numbers of false positives can skew the results of the analysis. Hence, in order to mitigate this issue, we take the following steps:

We create two meta categories of bug categories in FindBugs. The first category, which we call Primary category, includes bug categories like *Correctness*, *Multi-threaded Correctness*, *Performance*, and *Security*. FindBugs reports that the bug patterns reported in these categories are precise and actionable.

The second meta category, which we call Secondary category, includes categories such as *Bad Practice* and *Style*. Since FindBugs accepts more false positives in these categories, there are cases when one might decide that a bug pattern is not relevant for one's code base. For example, one never uses *Serialization* for persistent storage, so one never cares about the fact that one didn't define a *serializationUID*. But FindBugs would report this bug pattern anyway. Moreover, even for the bug patterns which are relevant to one's code base, perhaps only a minority will reflect problems serious enough to change the code. So in most cases, Secondary category does not pose a threat other than affecting the readability of code. Thus we conduct our analysis separately for Primary and Secondary category whenever relevant to avoid potential threats due to the nature of bug patterns the two categories detect.

Also, FindBugs assigns a severity level (HIGH, MEDIUM, or LOW) to each bug pattern based on its threat. To further mitigate the risk of false positives, we configured FindBugs to exclude LOW severity bug patterns. Examples of such low-severity bug patterns are *field names not starting with lower-case*, and *method that fails to close stream on exception*. The complete list of such bug patterns is available at FindBugs's website.

III. STUDY RESULTS

In this section, we present the results of our analyses of code clones and their relationship with a set of bug patterns reported by FindBugs. Our findings are mostly consistent across all the subject systems and we describe the places where they are different. Each sub-section below specifically addresses the research questions posed in Section I.

A. RQ1: Is defect density of cloned code greater than that of non-cloned code?

To answer this question, we compare the defect density of cloned code with non-cloned code across all the projects. Column 4 (Defect Density) in Table I shows the defect density of cloned and non-cloned code for each project respectively. We found that in 26 out of 31 projects, the defect density of cloned code is lower than that of non-cloned code.

As discussed, Primary and Secondary category differ in the severity and the type of bug patterns they detect. Secondary category consists of bug patterns related to *Style* and *Bad Practice*. In order to avoid the threat of such non-crucial bug

patterns impacting RQ1, we only consider Primary category for our analysis to answer RQ1. However, for the sake of completeness, we also describe the data for all the categories as well as separately for Primary and Secondary category.

Figure 2 shows three box plots comparing median defect density (Y axis) in cloned code and non-cloned code across three groups consisting of "All", "Primary", and "Secondary" bug category (l-r). As shown in boxplot (left), considering all the categories, the difference in the median defect densities of non-cloned (4.06) and cloned code (3.06) is 1. The difference decreases to almost 0 (0.02) when we consider only secondary bug-category (right boxplot). This implies that the observed difference in the median defect densities when considering all the categories is mainly because of the primary category - which is our category of interest because it consists of severe and more problematic bug patterns. For the primary category, the median defect density of the cloned code and non-cloned code is 0.58 and 2.14 respectively (center boxplot), making the defect density of cloned code 3.7 times less than that of non-cloned code.

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. We wanted to explore if the number of bug patterns in a method also follow a similar trend. Since the total size of cloned code is much smaller than that of non-cloned code, we considered the possibility that this size difference is serving as a confounding factor. Moreover, we found that the average method size of clones is 17 LoC and that of non-clones is 7 LoC. As such, the following two factors can impact the result of our analysis: (i) the relative difference in the average method size of cloned and non-cloned code for each project; and (ii) the relative difference in the total number of cloned and non-cloned methods for each project.

In order to understand the impact of method size on our analysis, we first compute the Pearson correlation coefficient between method size (LoC of a method) and the number of bug patterns found in a method. The goal is to determine how strongly, the number of bug patterns in a method correlate with the method size. Clearly, if the number of bug patterns show a strong positive correlation with the method size, then the code group (clone/non-clone) with bigger methods i.e., cloned code, is likely to have more bugs.

The correlation between bug patterns and method size for all the methods in the corpus is 0.151; 0.108 for only cloned methods, and 0.147 for only non-cloned methods. Figure 3 shows the scatter plots between bug patterns and method size for all the methods, only cloned methods, and only non-cloned methods (l-r) transformed on a log-scale. Manual inspection of the scatter-plots confirmed that the correlations fell near zero and represented pairings without an identifiable relationship. We found that the distribution of bug patterns in the project is right skewed, meaning that majority of the methods contain very few bug patterns. This result is not unexpected, and confirms the intuition that some methods have many bug patterns while most of the methods contain hardly any bug pattern at all. This heavy right skew also means that scatter plots between bug patterns and method size will have the majority of points clustered at low values if a log transformation is not done.

Column 5 (Correlation) in Table I shows the correlation coefficient for each project individually. As shown, most of the projects show very low value of correlation coefficient.

The fact that method size is very weakly correlated with the number of bug patterns found in that method, is an indication that method size may not be serving as a strong confounding factor. While the above experiment ensures that the difference in the average method size of clones and non-clones does not impact the result of RQ1, the large difference between the total number of cloned and non-cloned methods might still pose a threat to our analysis. In order to address this, we performed the following two experiments.

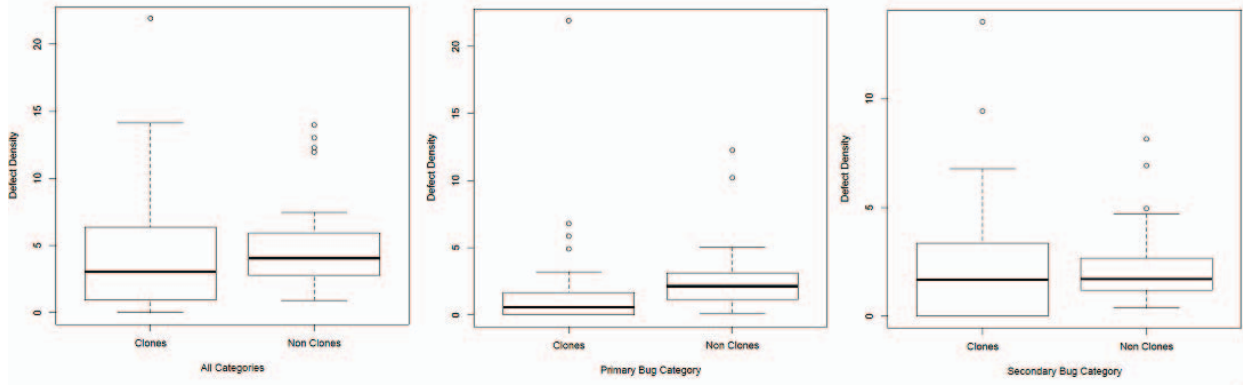


Fig. 2: Box plot showing defect density of cloned-code and non-cloned code. Note that defect density of cloned-code is less than that of non-cloned code for all the categories (left). For primary category, the defect density of cloned-code is 3.7 times less than non-cloned code (center). For secondary category the difference is zero (right), implying that most of the bugs in cloned code are of secondary category which consists of least problematic categories in FindBugs.

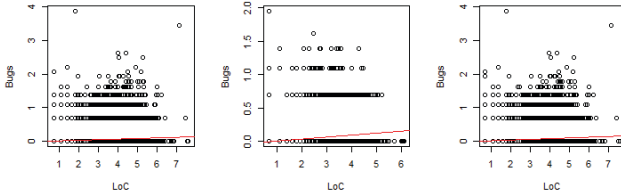


Fig. 3: Scatter plots of method size and bug patterns for only cloned, only non-cloned, and all the methods (left-right) reveal no identifiable relationship.

In the first experiment, for each project, we randomly pick, from the pool of non-cloned methods, only as many methods as the total cloned methods in the project. The goal is to have equal number of methods in both the groups and then compare their defect densities. We found that the average defect density of cloned code to be 4 times less (0.58 vs. 2.32) than that of non-cloned code.

In the above experiment, although, both the code groups have the same number of methods, because of the difference in the average size of method in each group (17 LoC vs. 7 LoC), the average LoC in each group may be different. Hence, in order to further ensure the validity of our analysis, we modify the above experiment to randomly pick non-cloned methods, one by one, such that the total LoC of non-cloned code group is same as that of cloned code group. Note that in this case, the total number of methods in each group may be different, but, each group will have same LoC. We found that, even under this setting, the defect density of cloned code is 3.1 times less (0.58 vs. 1.82) than that of non-cloned code.

Columns 6 (Defect Density (size control)) in Table I shows the defect density of non-cloned code for each project separately for each experiment. Note that the results are averaged over 100 runs to ensure sufficient randomization while picking the non-cloned methods. Below we show statistical significance of the difference.

Statistical significance of the difference. We use statistical paired tests to study the significance of the difference between median defect densities of the two groups. We formulate the hypotheses as follows:

$$H_0 : \mu(r_{NC} - r_C) \leq 0, H_A : \mu(r_{NC} - r_C) > 0$$

where $\mu(r_{NC} - r_C)$ is the population median of the difference in defect density between the non-cloned code (NC) and the cloned code (C) for each project.

H_0 : The median defect density of non-cloned code is less than or equal to that of the cloned code,

H_A : The median defect density of non-cloned is greater than that of cloned code.

For our analysis, we conduct Wilcoxon signed rank test because it is a non-parametric paired statistical test and does not make any assumptions about the normality of the data [39]. The computed p-value is 0.0031 (< 0.05). We note that the median defect density in the clone code is 0.58 and that of the non-cloned code is 2.14. Large scale studies are often guaranteed to give small p-values, therefore we are also reporting the summary statistics (medians), so that the reader can judge the significance of the differences.

The defect density computed using FindBugs in cloned code is 3.7 times less than that of the rest of the code, implying that there is a positive differentiation of cloned code with respect to the rest of the code

B. RQ2: Are there specific bug categories which are seen more often in the cloned code?

To answer this question, we examine the relationship between bug categories and cloned code. Since each category poses a different threat level, such analysis will be useful for risk assessment and employing targeted measures to mitigate the risks.

Figure 4 shows how bug patterns found in the cloned code are distributed across various categories. The number of bug patterns found in all the projects are on the Y-axis and bug categories are listed on the X-axis. As seen, 66% (399 out of 606) of bug patterns found in the cloned code belong to *Style* and *Bad Practice* categories (Secondary category). This number is as low as 49% (2065 out of 4206) for non-cloned code. It is worth noting that these bug patterns are mostly violations of recommended and essential coding practice and are susceptible to more false positives. This implies that not only cloned code have fewer bug patterns compared to non-cloned code, most of the bug patterns belong to the two least problematic category of FindBugs.

Statistical significance of the difference. In order to statistically evaluate the dominance of Secondary category in the cloned code, we formulate the following hypotheses:

$$H_0 : \mu(r_P - r_S) \geq 0, H_A : \mu(r_P - r_S) < 0$$

where $\mu(r_P - r_S)$ is the population median of the difference in bug pattern count between Primary category (P) and Secondary category (S) in cloned code.

Project	SLOC		#Method		Defect Density		Correlation	Defect Density (Non-clones) (size control)		Bugs	
	Total	Clones	Total	Clones	Non-clones	Clones		Equal # Methods	Equal LoC	duplicated	non-duplicated
ant	86,438	3,642	7,883	208	2.66	1.65	0.195	2.46	2.61	4	0
berkeleyparser	57,905	7,842	3,364	357	4.35	0.89	0.168	4.08	4.23	9	10
cglib	13,668	354	1,916	28	2.69	0	0.123	3.74	1.65	0	0
cloud9	56,766	21,066	4,390	1,047	11.96	6.5	0.149	5.02	5.07	–	–
cocoon	10,387	211	773	25	2.08	4.92	0.197	3.87	1.82	1	0
commons-io	8,673	440	674	50	1.05	0	0.137	2.97	3.56	1	1
dom4j	17,854	997	2,206	72	3.74	0	0.096	0.82	0.8	1	0
hadoop-hdfs	70,411	5,016	5,401	362	2.48	2.03	0.051	1.05	1	0	1
hadoop-mapred	64,023	3,452	5,577	230	3.05	1.28	0.205	2.4	2.25	2	1
hibernate	72,409	3,117	8,973	243	4.18	0	0.017	2.79	2.97	6	1
httpClient	18,022	1,206	1,324	102	3.06	6.8	0.101	1.6	1.81	2	0
jfreechart	93,460	15,470	7,034	871	1.01	0.2	0.214	0.89	0.72	89	31
jython	211,905	7,411	19,593	620	1.47	1.66	0	0.1	0.22	1	2
log4j	16,111	609	1,687	57	0.86	0.58	0.142	2.32	2.24	2	1
lucene	15,670	1,438	1,315	77	0.14	0.13	0.152	1.66	1.06	4	2
mahout-core	53,366	3,800	3,943	249	1.61	0	0.003	1.54	1.09	0	0
mason	35,931	3,800	2,674	261	1.48	0	0.278	1.37	1.36	8	5
nutch	12,243	519	861	40	4.24	0	0.299	4.49	4.34	1	0
pdfbox	13,936	1,653	894	31	12.27	0	0.256	14.81	9.74	0	0
pig	84,770	3,059	5,458	180	3.03	0.33	0.157	2.97	2.27	1	7
pmd	60,060	6,864	4,634	254	0.79	0.58	0.33	0.67	0.7	76	2
poi	47,804	5,093	4,899	343	4.87	2.16	0.117	4.8	4.18	6	9
postgresql	23,514	1,916	1,836	160	10.22	21.91	0.415	8.64	8.07	20	2
rhino	54,722	2,412	2,958	135	0.87	0	0.209	0.5	2.4	0	0
stanford-nlp	210,233	4,430	8,692	295	5.72	9.26	0.192	3.18	3.45	–	–
struts	24,799	1,275	2,300	82	1.61	0.78	0.127	1.87	1.24	0	1
substance	47,361	5,014	3,026	213	1.32	0.4	0.233	1.13	0.84	8	1
synapse-core	41,612	2,370	2,803	138	0.78	0	0.18	0.88	1	1	2
tomcat-catalina	73,673	4,030	4,757	232	2.32	0.99	0.171	2.26	2.76	5	2
uima-core	11,942	687	947	59	2.14	0	0.007	2.53	0.8	0	0
xerces	76,185	7,340	5,131	345	0.91	0.68	0.205	1	0.87	6	4

TABLE I: Results. Correlation column shows Pearson correlation coefficient between method size and # of bug patterns. Defect density (size control) shows defect density of non-cloned code when (i) using the same number of non-cloned methods as cloned code methods (Equal # Methods); and (ii) using non-cloned methods whose LoC sums up to total cloned method LoC (Equal LoC)

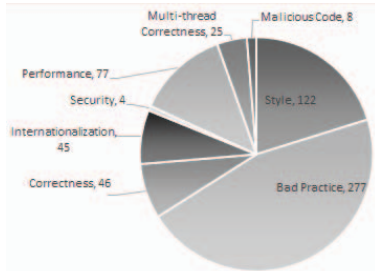


Fig. 4: Bug patterns in cloned code classified into various categories

H_0 : The median bug pattern count of Secondary category is greater than or equal to Primary category.

H_A : The median bug pattern count of Secondary category is less than that of Primary category. the p-value obtained using Wilcoxon signed rank test is 0.0415. While the median number of bug patterns found in both the primary and the secondary category is equal to 2, the mean number of bugs found in the primary and secondary category are 6.67 and 12.87 respectively.

66% of the bug patterns associated with code clones are related to issues in coding style and practice, the two least problematic of the FindBugs' categories, while that number is 49% for non-cloned code

C. RQ3: How often do bug-patterns propagate through cloning?

In order to investigate how often bug patterns are duplicated when the code is cloned, we manually examined 339 code clone groups which had at least one bug pattern present in them. Note that a code clone group is a set of methods which are clones of each other. For each code clone group, we look at all the bug patterns present in the group one at a time, and classify the group into one of the following two categories.

Category 1 - If the bug pattern present in the code clone is also present in at least one other member of the same code clone group. This implies that the bug pattern was also duplicated during code cloning.

Category 2 - If the bug pattern is unique to the code clone group i.e., it is present in only one member of the group. This implies that the bug pattern was either introduced later (after code cloning) to one of the instances, or it was fixed in all the other instances after cloning.

Column 10 & 11 in Table I show number of duplicated and non-duplicated bug patterns for 29 (out of 31) projects

respectively. In total, out of 339, we found that 254 code clone groups were classified in Category 1 and only 85 in Category 2.

75% of the FindBugs' bug patterns in cloned code are duplicated without any changes, while 25% are only present in one of the clones. Interpreting this result, while one may argue that cloning such a piece of code increased the total number of bug patterns, it is interesting and important to understand these results in the light of RQ2's results.

Most of these duplicated bugs again fall into coding and style category. On inspecting the code clone siblings with duplicated bug patterns, we found that it is very likely, that even without cloning, many developers would have written the code of same quality (i.e., with bug patterns present in them), if not worse. However, since bug patterns found in these categories are not very complex, a tool support to proactively assist developers make informed decision regarding cloning maybe very useful to maintain the code quality along with the advantages of rapid development using code cloning.

For example, we found that the two siblings of a clone group had identical code except hard coded integer values passed as parameter to the methods. Although this does not result in any serious threat to the program, it is definitely a bad practice because it breaks abstraction and increases the code size. In this case, an easy fix would be to refactor the two methods into one method by modifying the signature of any of these methods, so that it accepts an integer type as a parameter. Today, automated refactoring support for such scenarios often leaves the burden of identifying the refactoring candidates on the developers and hence it is often perceived as an optional exercise. A more proactive approach, for example, in the above case, automatically detecting candidate methods for merging by detecting clones as soon as a method is completed by a developer, and then showing the identified FindBugs' bug pattern (hard-coded integer), will help developer to not only develop faster but also incrementally clean the code base. Thus we posit that making developers aware of such issues and assisting them with code refinement while they are copy pasting will blend better with the development activity and hence it is likely be more adopted.

75% of the FindBugs' bug patterns in cloned code are duplicated without any changes, while 25% are only present in one of the clones. As most of the bug patterns duplicated come from Coding & Style bug category, this finding has important implications on the part of tool designers

D. Manual Inspection

Methodology. In order to gain more insights about why cloned code has lesser defect density than their non-clones counterpart, two of the authors manually inspected 100 clone groups found in 31 java projects. The number of clone siblings in these clone groups range from 2 to 5,473. The authors use the categories introduced by Kasper and Godfrey in [28] for classification.

Cross-cutting Concerns. Two different methods responsible for different tasks can look like clones when apart for implementing the required behavior they also implement the code for addressing similar aspects like logging, authentication, debugging etc. Usually the code that implements the required behaviour is small and a developer usually copies the whole method body from one such method and changes only the part of the code that affects the functionality and hence the two methods look structurally very similar. As a large part of such methods contain the statements that mostly are the function calls to logging, debugging and authentication modules etc., it often is a well tested code and therefore the net addition of bug-patterns while copying such code is usually less. The code snippets 1A and 1B in Figure 5 show one such clone group where the methods deal with two concerns, a) checking users permission and b) making the function call to putAttribute method. In both of the siblings the code makes a function call

to the putAttribute method only after checking if the current user belongs to a role that has the necessary permission to complete the action.

Code Generation. The methods that are auto-generated using some tool usually have similar structure and such clones are a result of a process and not to be modified by developers. Hence, as long as this code of conduct is followed, presence of such clones is not a threat. For example, we observed a clone group with 5,473 members in *PyArrayDerived.java* in *Jython* project. The code snippet 2 shown in Figure 5 is written for *rlshift* operator and is duplicated for all the operators creating clones. Similarly, we found clones of *initComponents()* method in *FormEditor.java* of *Eclipse-jdt core* project which gets regenerated by the constructor of *FormEditor*. In such cases, if the code generator produces bug free code, cloned code will have code of lesser defect density.

API/Library Protocols. We observed clones that resulted due to the constraints or style imposed by the frameworks. Cloning in such case is a result of the limited APIs made available by the framework and the sequence in which they are to be invoked to achieve the task. Such tasks are usually domain independent and hence they are part of various projects, e.g., adding items in the menu, calendar functionality, internationalization, etc. The sequence of function calls in such clones is usually free of any static bug-pattern resulting into the introduction of lesser bug-patterns. For example, we found a clone of *createActionComponent()* (see the code snippet 3 in Figure 5), a Factory method which creates the *JMenuItem* for each *Action* added to the *JMenu*. Similarly the *run* method in *edu.umd.cloud9.example.hits.AFormatterWG* and *edu.umd.cloud9.example.hits.HFormatterWG* have a series of similar function calls to create and configure the mapper, reducer and combiner for invoking map-reduce jobs.

Replicate and Specialize. While developing a solution to a problem, the developers sometimes copy a piece of code that has already been implemented to address a similar problem. However, this code may not be the exact solution, but a little modification is required to address the required problem. The resultant defect density of the clone code in such cases depends on the original code and we can't say how cloning in such cases affects the defect density of cloned code without temporal data. We noticed several such cases where the clone siblings that perform exactly opposite tasks, however, the methods are structurally very similar. For example, the code snippets 4A and 4B in Figure 5 show two methods that iterate over all the elements of a user defined list and invoke *contains* and *removeValue* API respectively. Undoubtedly, these two code blocks share most of the code, however, their existence in two separate well defined methods could be a thoughtful decision. Similarly, the code snippets 5A and 5B in Figure 5 show another such example of two methods which share most of the code, but are designed to achieve separation of concerns.

Exact or Near-Exact Copy. We observed several instances where the clone siblings have identical or almost identical code. Such cases of clones could arise due to laziness on programmer's part or because of not following good development practices. In some instances we found code comments by developers reflecting that they are fully aware of such clones and they intend to modify or refactor the resulting clone method in future. For example, we found that the two siblings of a clone group had identical code except hard coded integer values passed as parameter to the methods (6A & 6B in Figure 5). Although this does not result in any serious threat to the program, it is definitely a bad practice because it breaks abstraction and increases the code size. In this case, an easy fix would be to refactor the two methods into one method by modifying the signature of any of these methods, so that it accepts an integer type as a parameter. A tool support that pro-actively detects such a bug pattern can help developer fix the code before cloning. There exists automated refactoring support for such scenarios, however, such an approach is reactive and hence often ends up as an optional exercise. We conjecture that a more proactive approach of making developer

```

1A) public void processNestedTag(PutTag nestedTag) throws JspException {
    // Check role
    HttpServletRequest request = (HttpServletRequest) pageContext.getRequest();
    String role = nestedTag.getRole();
    if (role != null && !request.isUserInRole(role)) {
        // not allowed : skip attribute
        return;
    }
    putAttribute(nestedTag.getName(), nestedTag.getRealValue());
}
/*****

1B) public void putAttribute(PutListTag nestedTag) throws JspException {
    // Check role
    HttpServletRequest request = (HttpServletRequest) pageContext.getRequest();
    String role = nestedTag.getRole();
    if (role != null && !request.isUserInRole(role)) {
        // not allowed : skip attribute
        return;
    }
    putAttribute(nestedTag.getName(), nestedTag.getList());
}
/*****

2) public PyObject __rshift__(PyObject other) {
    PyType self_type = getType();
    PyObject impl = self_type.lookup("__rshift__");
    if (impl != null) {
        PyObject res = impl.__get__(this, self_type).__call__(other);
        if (res == Py.NotImplemented)
            return null;
        return res;
    }
    return super.__rshift__(other);
}
/*****

3) protected JMenuItem createActionComponent(Action a) {
    JMenuItem mi = new JMenuItem((String) a.getValue(Action.NAME),
        (Icon) a.getValue(Action.SMALL_ICON));
    protected PropertyChangeListener createActionPropertyChangeListener(Action a)
    {
        PropertyChangeListener pcl = createActionChangeListener(this);
        if (pcl == null) {
            pcl = super.createActionPropertyChangeListener(a);
        }
        return pcl;
    }
    mi.setHorizontalTextPosition(JButton.TRAILING);
    mi.setVerticalTextPosition(JButton.CENTER);
    mi.setEnabled(a.isEnabled());
    return mi;
}
/*****

4A) public boolean containsAll(final IntList c) {
    boolean rval = true;
    if (this != c) {
        for (int j = 0; rval && (j < c._limit; j++) {
            if (!contains(c._array[j])) {
                rval = false;
            }
        }
    }
    return rval;
}
/*****

4B) public boolean removeAll(final IntList c) {
    boolean rval = false;
    for (int j = 0; j < c._limit; j++) {
        if (removeValue(c._array[j])) {
            rval = true;
        }
    }
    return rval;
}
/*****

5A) public void startEntity(String name) throws SAXException {
    for (int i = 0; i < this.lexicalHandlerList.length; i++) {
        if (this.lexicalHandlerList[i] != null)
            this.lexicalHandlerList[i].startEntity(name);
    }
}
/*****

5B) public void endEntity(String name) throws SAXException {
    for (int i = 0; i < this.lexicalHandlerList.length; i++) {
        if (this.lexicalHandlerList[i] != null)
            this.lexicalHandlerList[i].endEntity(name);
    }
}
/*****

6A) public static String getAuthority(String uri) {
    RE re = RE(uripattern);
    if (re.match(uri)) {
        return re.getParen(4);
    } else {
        throw new IllegalArgumentException(uri + " is not a correct URI");
    }
}
/*****

6B) public static String getPath(String uri) {
    RE re = RE(uripattern);
    if (re.match(uri)) {
        return re.getParen(5);
    } else {
        throw new IllegalArgumentException(uri + "is not a correct URI");
    }
}
}

```

Fig. 5: Sample code clones observed in the subject systems. 1A & 1B: Cross-cutting Concerns; 2: Code Generation; 3: API/Library Protocols; 4A, 4B & 5A, 5B: Replicate and Specialize; 6A & 6B: Near-Exact Copy

aware of such issues while she is copy pasting will blend with the development activity and hence will be more adopted.

Overall, in all of the observed clone groups, we came to an understanding that the developers seemed to have copied code, which is sufficiently well written and handle the edge cases. Cases where cloning also resulted into duplication of bug patterns and hence impacted overall code quality, could have been fixed by adding proactive tool support to help developers make an informed decision. In their study [54], Thummalapenta et al. also found that clones were consistently propagated when needed and developers actually seem to remember the clone locations that require such propagation, particularly in important cases like bug fixes. This suggests that although there may be many inconsistent changes, developers are aware and make conscious decision about propagating the change.

IV. THREATS TO VALIDITY

In this section, we identify the following threats to the validity of our research.

Robustness of Clone Detection Technique. We classify methods as clones or nonclones. However, this classification is as accurate as the choice of our clone detection technique, tool, and configurations [57]. In order to mitigate this risk, we chose our own tool, with which we have previous experience [51]. Thus helping us to choose the right configurations for better accuracy. Moreover we manually verified random samples and found false positives to be under 5%. We plan to use the output of one more clone detection tool and hence further lower the risk of being biased toward any specific tool results. Moreover, we only consider method-level clones. So we may miss overlapping clones or clones in class definitions. However, given this study is for Java projects, we believe that most of the clones representing logical blocks of program should be captured at a method level. Nonetheless, we acknowledge that this assumption might still impact the study.

False Positives in Bug Patterns. The imprecise nature of bug patterns reported by FindBugs is another threat to our validity. To address this issue, we only consider bug categories which have proven to be precise and actionable. Moreover, we exclude bugs with LOW severity. Nonetheless, it is very likely that we still face the issue of false positives. Moreover, although FindBugs detects a variety of bug patterns across various categories, there may be other bug patterns which are not detected. So the results of this study are to be interpreted only in the context of FindBugs' bug patterns.

Use of FindBugs in The Development Practice. It is possible that developers were already using FindBugs during development, and the warnings were fixed before release, and thus fewer post-release defects would be associated with warnings. This would further reduce the defect density and skew the analysis. We took a random sample of 20 projects, and found no evidence of systematic use of FindBugs tool. We did this by going through the developer fixes, and examining the source history of these projects. In addition, we found no evidence in the email archives of any these projects suggesting a systematic adoption of FindBugs tool. These observations provide some mitigation to this particular threat.

Generalizability. The results of this study are from 31 open source Java systems, all medium to large size. We chose subject systems that exhibit variety in their type, size, and domain to minimize the impact of such factors on the observed phenomena. However, drawing general conclusions from empirical studies in software engineering is difficult because besides independent variables, the process depends on many relevant confounding variables [4].

For example, Mondal et al. [41] found that old clones are stable. Does stability of clones make them less bug prone? If yes, clone age could be one such confounding factor that needs to be controlled. Similarly, criticality of clones could play an

important role i.e., are clones present in all the features or just not-so-critical features? This will validate that cloned code has fewer bug patterns not just because it does not implement a critical feature of the system, but because of its peculiarity. This can be done by preserving the homogeneity of cloned and non-cloned methods, which is currently not accounted for.

Thus, in the presence of such confounding factors, we cannot assume *a priori* that the results of the 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 systems and indicate that the results hold for more than just the studied systems.

V. RELATED WORK

Code cloning has a long history as a topic of inquiry in the research community. Although there are many negative effects of code cloning [24], [16], [38], [37], [58], [26], the widespread presence of clones have motivated researchers to dig deeper and understand the usage scenarios.

Kasper et al. [28] presented eleven patterns by examining clones in two systems. They found out that not all usage patterns have negative consequence and some may even have positive consequence on quality.

Ossher et al. [44] looked at circumstances of file cloning in open source Java systems and classified the cloning scenarios into good, bad and ugly. 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 subprojects. From a software engineering standpoint, some of these situations are more legitimate than others.

Kim et al. [29] studied clone evolution in two open source systems. 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. 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. To that end, de Wit et al. [13] proposed CLONEBOARD, an Eclipse plug-in implementation to track live changes to clones and offering several resolution strategies for inconsistently modified clones. 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 [10] analyzes clones and intentions behind cloning of a financial institution system and argues that external business factors may facilitate cloning. He mentions 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. [48] found that reducing duplication in a web application only had negative effects on the modifiability of an application. He notes that after significantly reducing the size of the source code a single change required testing of a vastly larger portion of the system.

Rahman et al. [46] investigate the effect of cloning on defect proneness on four open source systems. 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 supports our findings as well.

Brutnik et al. [8] use clone detection techniques in a novel way to find cross-cutting concerns in the code. They manually identify five specific cross-cutting concerns in an industrial C system 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 a set of bug patterns related to code clones, which were not previously investigated by other studies. These bug patterns 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 system, 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. We conduct this study on 31 open source Java systems. To the best of our knowledge, our study is the first to perform clone-quality analysis at this level. Since FindBugs patterns have high level categorization, we establish association between clones and the most crucial bug categories. This not only helps in analyzing the risk associated with cloning, but could also be useful to figure plan out testing and defect fixing of a method.

VI. CONCLUSION

We conducted an empirical study of 31 open source Java projects (1.7 MSLOC) to explore the relationship between code clones and a set of bug patterns reported by FindBugs and found that: (i) the defect density in cloned code is 3.7 times less than that of the rest of the code; (ii) 66% of the bug patterns associated with code clones are related to issues in coding style and practice, the two least problematic of the FindBugs' categories, while that number is 49% for non-cloned code; and (iii) 75% of the bug patterns in cloned code are duplicated without any changes, while 25% are only present in one of the clones.

Our findings resonate with some of the recent studies, and provide one more piece of evidence that cloning may not always be harmful. All of these studies including ours, suggest that the issue of code cloning is a complex one and there are several pros and cons to it. While code cloning has its side effects - code bloat, inconsistent changes, breaking abstractions, etc., it is also evident that the practice of code cloning has become an integral part of our software development process. Perhaps it is an opportunity to focus on tools and techniques that help developers take advantage of rapid development using cloning and also manage clones effectively (and automatically) to counter the known side effects of code cloning.

Reproducibility. While our results appear to be statistically significant, we urge caution in extending the findings to other languages. Comparisons such as these are key to promoting software engineering discipline in understanding code clone properties and we invite others to use our dataset for further experiments. We have made available all the necessary artifacts including project sources, detailed steps to run tools and produce the raw data, analysis steps to produce the statistical results to verify the claims at: <http://mondego.ics.uci.edu/projects/bugpatternsinclones/>

Acknowledgements. This material is based upon work supported by the National Science Foundation under Grant No. 1018374.

REFERENCES

- [1] N. Ayewah and W. Pugh. Using findbugs on production software. In *OOPSLA'07: Companion to the 22nd ACM SIGPLAN conference on Object oriented programming systems and applications companion*, pages 805–806. ACM, 2007.
- [2] B. Baker. A program for identifying duplicated code. *Computing Science and Statistics*, pages 24–49, 1992.
- [3] B. S. Baker. On finding duplication and near-duplication in large software systems. In *Proceedings of Working Conference on Reverse Engineering*, 1995.

- [4] V. Basili, F. Shull, and F. Lanubile. Building knowledge through families of experiments. *Software Engineering, IEEE Transactions on*, 25(4):456–473, 1999.
- [5] 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*, page 368. IEEE Computer Society, 1998.
- [6] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, and E. Merlo. Comparison and Evaluation of Clone Detection Tools. *IEEE Trans. Softw. Eng.*, 33(9):577–591, 2007.
- [7] C. Bird, A. Bachmann, E. Aune, J. Duffy, A. Bernstein, V. Filkov, and P. Devanbu. Fair and Balanced? Bias in Bug-Fix Datasets. In *Proceedings of the the Seventh joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, 2009.
- [8] 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*, 31(10):804–818, 2005.
- [9] Clone detector. <http://mondego.ics.uci.edu/projects/clonedetection/>.
- [10] J. Cordy. Comprehending reality - practical barriers to industrial adoption of software maintenance automation. In *Proceedings of International Conference on Program Comprehension*, pages 196–205, 2003.
- [11] J. Cordy and C. Roy. The nicad clone detector. In *Proceedings of ICPC*, 2011.
- [12] Coverity scan. <http://scan.coverity.com/>.
- [13] 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.
- [14] 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*, page 109. IEEE Computer Society, 1999.
- [15] Findbugs. <http://findbugs.sourceforge.net/>.
- [16] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts. *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 1999.
- [17] N. Gode and R. Koschke. Incremental clone detection. In *Proceedings of CSMR*, 2009.
- [18] I. Herraiz, E. Shihab, T. H. Nguyen, and A. E. Hassan. Impact of installation counts on perceived quality: A case study on Debian. In *Proceedings of the 18th Working Conference on Reverse Engineering*. IEEE Computer Society, 2011.
- [19] K. Herzig, S. Just, and A. Zeller. It’s not a bug, it’s a feature: How misclassification impacts bug prediction. In *Proceedings of the 2013 International Conference on Software Engineering, ICSE ’13*, pages 392–401, Piscataway, NJ, USA, 2013. IEEE Press.
- [20] D. Hovemeyer and W. Pugh. Finding bugs is easy. *SIGPLAN Not.*, 39(12):92–106, Dec. 2004.
- [21] Y. Jia, B. Binkley, M. Harman, J. Krinke, and M. Matsushita. A proposed approach to fast and precise clone detection. In *Proceedings of IWSC*, 2009.
- [22] L. Jiang, G. Mishserghi, Z. Su, and S. Glondu. Deckard: Scalable and accurate tree-based detection of code clones. In *Proceedings of ICSE*, 2007.
- [23] 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*, pages 171–183, Toronto, Ontario, Canada, 1993. IBM Press.
- [24] J. H. Johnson. Substring matching for clone detection and change tracking. In *International Conference on Software Maintenance*, pages 120–126, 1994.
- [25] E. Juergens, F. Deissenboeck, and B. Hummel. Clonedetective: a workbench for clone detection research. In *Proceedings of ICSE*, 2009.
- [26] E. Juergens, F. Deissenboeck, B. Hummel, and S. Wagner. Do code clones matter? In *Proceedings of ICSE*, pages 485–495, 2009.
- [27] 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.*, 28(7):654–670, 2002.
- [28] C. Kapser and M. Godfrey. “cloning considered harmful” considered harmful: patterns of cloning in software. *Empirical Software Engineering*, 13(6):645–692, 2008.
- [29] M. Kim, V. Sazawal, D. Notkin, and G. Murphy. An empirical study of code clone genealogies. In *Proceedings of FSE*, 2005.
- [30] R. Komondoor and S. Horwitz. Using Slicing to Identify Duplication in Source Code. In *Proceedings of the 8th International Symposium on Static Analysis*, pages 40–56. Springer-Verlag, 2001.
- [31] R. Komondoor and S. Horwitz. Effective automatic procedure extraction. In *Proceedings of the International Workshop on Program Comprehension*, pages 40–56. Springer-Verlag, 2003.
- [32] R. Koschke. Survey of research on software clones. In *Proceedings of Duplication, Redundancy, and Similarity in Software*, 2007.
- [33] R. Koschke. Identifying and removing software clones. pages 15–36, 2008.
- [34] R. Koschke. Large-scale inter-system clone detection using suffix trees. In *Proceedings of CSMR*, pages 309–318, 2012.
- [35] R. Koschke, R. Falke, and P. Frenzel. Clone detection using abstract syntax suffix trees. In *Working Conference on Reverse Engineering (WCRE’06)*. IEEE Computer Society, 2006.
- [36] J. Krinke. Identifying Similar Code with Program Dependence Graphs. In *Proceedings of the Eighth Working Conference on Reverse Engineering (WCRE’01)*, page 301. IEEE Computer Society, 2001.
- [37] J. Krinke. A study of consistent and inconsistent changes to code clones. In *Working Conference on Reverse Engineering*, pages 170–178, 2007.
- [38] A. Lozano, M. Wermelinger, and B. Nuseibeh. Evaluating the harmfulness of cloning: a change based experiment. In *Mining Software Repositories*, pages 18–22, 2007.
- [39] K. M., N. C., N. J., and L. W. *Applied Linear Statistical Models*. Sage Publications.
- [40] J. Mayrand, C. Leblanc, and E. Merlo. Experiment on the automatic detection of function clones in a software system using metrics. In *ICSM*. IEEE, 1996.
- [41] M. Mondal, C. K. Roy, M. S. Rahman, R. K. Saha, J. Krinke, and K. A. Schneider. Comparative stability of cloned and non-cloned code: An empirical study. In *Proceedings of the 27th Annual ACM Symposium on Applied Computing, SAC ’12*, pages 1227–1234, New York, NY, USA, 2012. ACM.
- [42] K. S. Muhammad Asaduzzaman, C. K. Roy and M. D. Penta. Lhdiff: A language-independent hybrid approach for tracking source code lines. In *Proceedings of International Conference on Software Maintenance*. IEEE Computer Society, 2013.
- [43] T. T. Nguyen, H. A. Nguyen, J. M. Al-Kofahi, N. H. Pham, and T. N. Nguyen. Scalable and incremental clone detection for evolving software. In *Proceedings of International Conference on Software Maintenance*. IEEE Computer Society, 2009.
- [44] J. Ossher, H. Sajani, and C. V. Lopes. File cloning in open source java projects: The good, the bad, and the ugly. In *ICSM*. IEEE, 2011.
- [45] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Commun. ACM*, 15(12):1053–1058, Dec. 1972.
- [46] F. Rahman, C. Bird, and P. Devanbu. Clones: what is that smell? *Empirical Software Engineering*, 17(4-5):503–530, 2012.
- [47] F. Rahman, S. Khatri, E. T. Barr, and P. Devanbu. Comparing Static Bug Finders and Statistical Prediction. In *Proceedings of the International Conference on Software Engineering*, Hyderabad, India, 2014. ACM.
- [48] 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*, pages 116–126, 2007.
- [49] C. K. Roy and J. R. Cordy. A survey on software clone detection research. *Technical Report, Queen’s University at Kingston*, 2007.
- [50] 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*, pages 470–495, 2009.
- [51] H. Sajani and C. Lopes. A parallel and efficient approach to large scale code clone detection. In *Proceedings of International Workshop on Software Clones*, 2013.
- [52] H. Sajani, V. Saini, J. Ossher, and C. Lopes. Is popularity a measure of its quality? an analysis of maven components. In *Proceedings of the 30th International Conference on Software Maintenance and Evolution (To appear in ICSME 2014)*. IEEE Computer Society, 2014.
- [53] M. Shomrat and Y. Feldman. Detecting refactored clones. In G. Castagna, editor, *ECOOP 2013 European Conference on Object-Oriented Programming*, volume 7920 of *Lecture Notes in Computer Science*, pages 502–526. Springer Berlin Heidelberg, 2013.
- [54] T. Suresh, C. Luigi, A. Lerina, and D. P. Massimiliano. An empirical study on the maintenance of source code clones. *Empirical Software Engineering*, 15(1):1–34, 2010.
- [55] M. Toomim, A. Begel, and S. Graham. Managing duplicated code with linked editing. In *Visual Languages and Human Centric Computing, 2004 IEEE Symposium on*, pages 173–180, 2004.
- [56] S. Uddin, C. Roy, K. Schneider, and A. Hindle. On the effectiveness of simhash for detecting near-miss clones in large scale software systems. In *Proceedings of Working Conference on Reverse Engineering*, 2011.
- [57] T. Wang, M. Harman, Y. Jia, and J. Krinke. Searching for better configurations: A rigorous approach to clone evaluation. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2013*, pages 455–465, New York, NY, USA, 2013. ACM.
- [58] 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*, pages 94–103. IEEE, 2014.