Is Popularity a Measure of Quality? 
An Analysis of Maven Components

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

Abstract—One of the perceived values of open source software is the idea that many eyes can increase code quality and reduce the amount of bugs. This perception, however, has been questioned by some due the lack of supporting evidence.

This paper presents an empirical analysis focusing on the relationship between the utilization of open source components and their engineering quality. In this study, we determine the popularity of 2,406 Maven components by calculating their usage across 55,191 open source Java projects. As a proxy of code quality for a component, we calculate (i) its defect density using the set of bug patterns reported by FindBugs; and (ii) 9 popular software quality metrics from the SQO-OSS quality model. We then look for correlations between (i) popularity and defect density; and (ii) popularity and software quality metrics.

In most of the cases, no correlations were found. In cases where minor correlations exist, they are driven by component size. Statistically speaking, and using the methods in this study, the Maven repository does not seem to support the “many eyeballs” effect. We conjecture that the utilization of open source components is driven by factors other than their engineering quality, an interpretation that is supported by the findings in this study.

I. INTRODUCTION

“Given enough eyeballs all bugs are shallow” [1]. This quote is part of a number of beliefs about open source that many people take for granted. Some researchers, however, have questioned the validity of this belief. Robert Glass refers to it as a fallacy that lacks supporting evidence and that goes against the findings unveiled by research focusing on code reviews [2].

The research community has proposed and evaluated a number of beliefs about open source that many people take for granted. Robert Glass refers to it as a fallacy that lacks supporting evidence and that goes against the findings unveiled by research focusing on code reviews [2]. The validity of this belief has, indeed, never been directly tested. The recent Heartbleed bug (http://heartbleed.com/) found in one of the most widely used open source components has brought the popular belief about “many eyes” once again into question.

In another, unrelated, line of inquiry, software engineering researchers have studied the practice of open source component use, both in a descriptive and prescriptive manner. Researchers have explored real-world developer component selection behavior through the use of questionnaires and interviews [3], [4]. They have also proposed numerous assessment methodologies to guide developers in selecting components to reuse [5], [6], [7]. These empirical studies and component selection methodologies all suggest that component quality is, or should be, a central deciding factor in component selection. In general, these studies agree that quality should be subsidiary only to functionality requirements and license considerations when selecting a component. Despite this consensus, little work has been done to empirically study how component quality actually relates to component use in the real world.

Our goal with this study is to empirically analyze the relationship between component popularity and component quality at a large scale. As a proxy of code quality, we calculate defect density using the set of bug patterns reported by FindBugs, and 9 popular software quality metrics from the SQO-OSS quality model [8]. More specifically, we seek answers to the following research questions:

Research Question 1: Is there an inverse correlation between component popularity and its defect density?

In other words, do the most popular components have lower defect density than the less popular components? 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.

Using static analysis and pattern matching, FindBugs seeks to find defects using algorithms that process well-defined semantic abstractions of code. It is a proactive way of finding patterns which are potential threats for the code base, and hence such patterns constitute a direct code quality measure. Moreover, since large scale studies like ours are heuristic in nature, static defect finders like FindBugs provide necessary automation needed to perform the analysis on such large datasets. We discuss more about the rationale of using FindBugs’ bug patterns in the context of this study in 2.3.

Research Question 2: Do highly popular components have better values of software quality metrics than their less popular counterparts?

Code quality being a difficult concept to measure, it needs to be looked at from various dimensions. Thus, apart from computing defect density, we also include 9 SQO-OSS code quality metrics in our study.

The research community has proposed and evaluated a large number of software quality metrics. In general, these quality metrics are focused on assessing the complexity of a piece of software, with the belief that more complex software is more prone to defects. Complexity measuring metrics can be as simple as counting the lines of code, or as complex as counting the number of incident edges on a global dependency graph. Software quality metrics provide an indirect measure of finding patterns which are potential threats for the code. It is a proactive way of finding patterns which are potential threats for the code base, and hence such patterns constitute a direct code quality measure. Moreover, since large scale studies like ours are heuristic in nature, static defect finders like FindBugs provide necessary automation needed to perform the analysis on such large datasets. We discuss more about the rationale of using FindBugs’ bug patterns in the context of this study in 2.3.
supports their usefulness in specific situations [9], [10], [11], [12], [13].

While FindBugs attempts to identify certain types of common programming errors that can cause bugs, these metrics try to quantitatively assess the complexity of code which might indirectly lead to bugs.

Clearly, if we were to find that widely used components had lower defect density or scored higher on the software metrics, such findings would be a great boost to the popular belief underlying open source development.

Findings. In general, we found very few correlations, positive or inverse, between component popularity and quality, and the few correlations found were eliminated when component size was taken into consideration. This doesn’t mean that such correlations don’t exist; it simply means that we were unable to find them using the methods in our study. Statistically speaking, the results in this paper don’t support one of the most popular beliefs about open source software: specifically, the large Maven repository doesn’t seem to reflect the “many eyeballs” effect.

II. DATASETS

We use two different datasets for this empirical study. The primary dataset consists of Java artifacts hosted by Maven. The second dataset consists of open source Java projects from the Sourcerer repository [14]. We cross-reference the usage of the Maven artifacts in Sourcerer projects to compute the popularity of the Maven artifacts. We determine the quality of Maven artifacts by the use of static analysis for extracting defect density and software quality metrics, and then explore the relationship between quality and popularity.

A. Maven Components

Artifact repositories are popular library management solutions, as they provide a framework for collecting and organizing external artifacts. These repositories have a hierarchical structure where, at the top level, they contain libraries, such as JUnit. Each library is broken into multiple versions, such as JUnit4.9 and JUnit4.10. We used the Maven Central Repository, an example of such artifact repository, to download the subject systems. In Maven, libraries are called projects, and each project may have many versions. In total, we downloaded 64,054 artifacts from Maven. We eliminated Maven artifacts that were not used (zero usage) at all by the projects in the Sourcerer repository. Hence, out of 64,054, only a total of 17,243 artifacts were used by at least one project from the Sourcerer repository. Again, these 17,243 artifacts are specific versions of projects. For example, JUnit hosted on Maven, has 20 different versions (e.g., 3.7, 3.8, ..., 4.11). The collection of versions of a project logically represents a component. We refer to such collection of artifacts as an aggregated component. Thus, in the case of JUnit, this aggregated component is a collection of all of its 20 versions. We obtained 2,563 such aggregated components from a total of 17,243 versioned Maven artifacts. For the rest of the paper, we will refer to these aggregated components as components. All metrics reported in this study pertain to the aggregation, not to individual versions.

1http://mvnrepository.com/artifact/junit/junit

Figure 1 shows the distribution of 2,563 components. The x-axis show component sizes, as measured by metric FindBugs size (which is similar to non-whitespace lines of code), binned logarithmically. The y-axis shows the number of components falling within that bin. As seen in the left most column, there are 156 components which are less than 100 LOC. We remove them from our analysis as most of these components were either non-java or without any functionality. Thus our final set consists of 2,406 Maven components.

Figure 2 shows similar distribution but the size of the component is measured in number of filtered Java files present in the component. The purpose of Java file filtering is to eliminate duplicate files within the component. The filtering process identifies Java files that declare types with identical fully-qualified names, and resolves each conflict to include only one of the duplicate files. Often projects contain such duplicate files due to the developers testing out alternatives implementations or supporting multiple platforms. Typically, the project’s build system would manage these files, making sure that no naming conflicts would occur. Sourcerer’s filtering algorithm attempts to approximate this, eliminating naming conflicts in feature extraction. As a result of this duplication, the number of filtered files better approximates the true size of a component.

B. Sourcerer Projects

The latest version of the Sourcerer Repository was created specifically for the purpose of the study presented in this paper. It contains open source Java projects from three different sources: Google Project Hosting, Apache, and SourceForge.

Table I contains general statistics on the projects in the Sourcerer Repository. Each row summarizes the projects origi-
inventing from the specified internet repository, with the final row containing statistics on the totality of projects in the repository. The second column, projects, shows the number of projects downloaded from the source listed in the first column. As this table shows, Google Project Hosting is the largest contributor of projects to the Sourcerer Repository, with SourceForge in a close second place and Apache in a very distant third place. The third column contains the number of filtered Java files present in each group of projects.

These numbers clearly show that projects from Google Project Hosting are generally smaller than those in SourceForge, which in turn are smaller than those in Apache. This difference highlights the variation present within the open source community. While there are likely many factors causing this difference, we believe differences in marketing between the repositories accounts for much of the variation. Apache only includes projects by invitation, containing only those that align with the foundations goals and are of interest to a large group of developers. This results in a repository filled with large, mature projects. SourceForge, on the other hand, is one of the original repositories for hosting open source projects, and is generally associated with the more traditional open source community. Google Project Hosting, on the other hand, has more appeal among students and individual developers looking for places to host their projects, which has likely caused a skew towards small disposable projects.

III. STUDY DESIGN

As shown in Figure 3, the overall design of our study is a two-step process. In the first step, we compute the popularity of 2,406 Maven components across all the 55,191 non-empty Java projects. We describe this in detail below. At the end of this step, we have all the projects from the Sourcerer repository which use a given Maven component. In the second step, for each Maven component, we: (i) detect all the bug patterns present in the component using FindBugs; and (ii) compute all the metrics listed in Table II. Combining the result of the previous two steps, we have, for each Maven component: (1) its usage across 55,191 open source Java projects; and (2) the bug patterns and other metrics associated with the component.

A. Calculating Component Popularity

Component popularity can be measured in a number of ways. Most intuitively, perhaps, a component’s popularity can be defined in terms of that component’s actual use. When using a component, developers make use of the component’s public application-programming interface (API). In quantifying a component’s popularity, therefore, we can simply record how many, and in what manner, third-party projects make use of the component’s public API. This general approach of measuring popularity has been commonly used to study structured ecosystems such as Linux [15], [16] or Unix [17].

We statically compute the usage of maven components by looking at their consumers, the Sourcerer repository projects. We build a global dependency graph of the Sourcerer projects using static analysis to resolve type and method references of the Sourcerer projects to their external referents. We then identify that how many of the external references are to the types in Maven components. Using this information, we calculate the following three metrics to compute the usage of a given Maven component:

Projects using Component: Counts the number of projects using the component. This corresponds to the number of different projects using a type from a component. More specifically, a component is considered to be used by a project if that project references a type (fully qualified name) found within that component. If a project references multiple types of the same component, the usage is still counted as one. However, if there are multiple projects that refer to types found within that component, the usage of that component is equal to the number of unique projects that refer to the types within the given component.

Files using Component: Same as above, except counts the number of files using a component. This corresponds to the number of files using a type from the component.

Uses of Component: Unlike the above measures, this counts the number of times types or methods within the component are used. This corresponds to the number of times any type or method from within the component is referenced.

We believe that the above component utilization metrics are ideal for large-scale analysis because they are a direct measure of utilization across the population of studied projects. Moreover, the criterion is very broad, and does not risk discounting any legitimate uses. They are also relatively easy to compute, as it only requires matching missing type name against a collection of components. In addition, these measures can be computed from a component in isolation, without trying to link its binaries to their originating community.

Details on building the global dependency graph. In order to build the global dependency graph of Sourcerer projects, we start by building an abstract syntax tree (AST) and then attributing each type or method node with resolved binding information. This step is identical to what Java compilers do before lowering the AST to bytecode. As a result, Sourcerer uses Eclipse’s Java Development Toolkit to build the attributed AST, and then crawls over the resulting tree and exports the reference information [14]. When analyzing bytecode, however, the process is even simpler. One feature of Java bytecode is that all references are already fully qualified. Therefore one can simply iterate through the bytecode to export all referenced types.

B. Component Quality

Software quality is a difficult concept to precisely define and measure [18]. In McConnell’s seminal book, Code Complete [19], quality is divided into two major categories: internal quality and external quality. Internal quality is the quality of the code as apparent to the developers themselves,
including concepts such as maintainability, extensibility and understandability. External quality is the quality of the software as perceived by its consumers, including concepts such as correctness, performance, security and reliability.

Measuring external quality can be exceedingly challenging, as it requires identifying defects and inaccuracies in software. Such quality measurements are usually done by either using issue reports or by using static analysis tools. While both techniques have their merits and disadvantages [20], in the context of this study, there are several reasons why measuring external quality using bug reports would be inappropriate.

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 trackers, rather than in their own; and vice-versa [21]. This raises severe issues regarding completeness of bug 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. [22] 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, the number of reported bugs depends on the life and popularity of the project. A very old project may have thousands of reported bugs, while a project that has just started may have fewer bugs reported, if any. Also, popular projects tend to have a much higher rate of issue reports than less popular projects - also know as popularity bias. This simple fact would render any correlation analyses useless, as it tangles the independent variable (quality) with the dependent one (popularity). One key difference is that as opposed to using bug reports, defects reported by static bug finders like FindBugs may not be subject to such popularity bias. For these reasons, we focused our study on properties of the code itself, rather than of the artifacts produced by the community around the projects. In this context, it is worth noting that Israel et al. [23], measured the popularity of Debian packages by counting how many times a package was installed by users and correlated it with defect density computed using defects reported and/or fixed by users. Unlike as reported in this study, they didn’t find no correlation, but found that only very popular packages have a high defect density. Such varied findings demonstrate that the issue of quality is a complex one, and exploring various dimensions using different methodologies might help us to better understand the relation between popularity and quality.

FindBugs as a Proxy of Quality

In order to avoid threats resulting from the aforementioned problems with issue reports, we use FindBugs [24], a popular static analysis tool used heavily in both research and industry to find bugs. FindBugs uses 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 [25], [26]. 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 [20]. Hence bug patterns detected can therefore serve as an indirect measure of external quality, subject only to their limitations in actually detecting bugs.

FindBugs is an heuristic tool; as such, it suffers from both false positives and false negatives. False positives are particularly problematic. A previous study has shown that FindBugs results in slightly less than 50% false positives [26], which is a high rate. Nevertheless, we believe that FindBugs can serve as an approximation of code quality for the purposes of this study for the following reasons.
First, the rate of false positives in FindBugs affects all the components in about the same way. Since our study is correlation-al on a large dataset, 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 [27], 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 results. 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 will fix.²

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.

Reducing False Positives. As discussed above, 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 [26], we believe that large number of false positives can skew the results of the analysis. FindBugs assigns a priority level (HIGH, LOW) to each bug pattern based on their how confident it is about the bug pattern resulting in a bug. Thus to further limit the rate of false positives, we configured FindBugs to find HIGH and LOW priority bug patterns separately. We then compute defect density using HIGH + LOW priority bug patterns, only HIGH priority bug patterns, and only LOW priority bug patterns and find their correlation with popularity separately for each configuration.

SQO-OSS Metrics as a Proxy of Quality

To give another angle to the components’ quality assessment, we compute software quality metrics for them selected from the SQO-OSS quality model [8]. SQO-OSS was chosen because it was specifically designed for evaluating the quality of open source projects. SQO-OSS divides quality into two main categories, product (code) quality and community quality. Product quality is further broken down into maintainability, reliability and security. In total, the SQO-OSS model contains 10 different quality attributes and 37 unique metrics. Out of 37 total number of metrics, 22 can be computed from the source code. This eliminates all quality metrics from the community quality and reliability categories. We implemented a popular subset of following 9 metrics out of these 22 (see also Table II).

Efferent Coupling: Efferent coupling of a given type is the number of types referenced by that type. The general belief is that the types showing high levels of efferent coupling are more likely to be faulty as they depend on a larger number of types within the system [9], [28].

Afferent Coupling: It measures the total number of types referencing the given type. The general belief is that the highly used types within a system exhibits lower fault rates, as they have greater opportunity to be tested. So more the afferent coupling, the better quality the project has.

Lack of Cohesion: The goal of this metric is to measure the degree of interconnectedness of the elements of a class. Classes with low cohesion are thought to be poorly designed, as they contain a number of unrelated methods.

Depth of Inheritance: The metric is defined as the depth of the given class in the inheritance tree. In other words, it is the number of classes above a given class in the type hierarchy. General wisdom regarding object-oriented programming states that the class hierarchy should ideally be a forest rather than a single giant tree. Therefore, it is thought that classes deep in the inheritance hierarchy are likely to be more fault-prone, as they inherit more functionality from a larger number of sources. This supposition has been supported by previous research [9].

Ratio of Derived to Base Classes: It is the Number of Derived Classes divided by the Number of Base Classes. This metric gives a sense of how heavily type inheritance is used within the artifact. A class is considered to be a derived class if it directly extends a class that is not java.lang.Object. A base class is a class that directly extends java.lang.Object

Cyclomatic Complexity: The goal of Cyclomatic Complexity is to measure the number of possible control flow paths through a method. Higher values indicate more complex logic, which suggests that the code is harder to understand and maintain, and therefore of lower quality.

Weighted method per class: It uses the Cyclomatic Complexity for each method within a class to arrive at an overall complexity for the entire class. The idea is that while counting the number of methods within a class gives some idea of that class’ complexity, weighting each method by its Cyclomatic Complexity gives a more accurate measure.

Class comment frequency: The Class Comment Frequency metric measures the frequency of Javadoc comments within a block of code. It is equal to the Lines of Class Comments divided by the Non-Whitespace Lines of Code.

Vocabulary Frequency: The Vocabulary Frequency metric normalizes the Number of Instructions in a block of code by the size of the code’s vocabulary. The value of this metric is the number of instructions divided by the vocabulary size.

²Information published on FindBugs website: http://FindBugs.sourceforge.net/
The goal of our investigation is to empirically evaluate the general belief that highly utilized components are of higher quality than their less used counterparts. First, in order to properly evaluate the relationship between the utilization metrics and the quality metrics we need to understand the distribution of the utilization metric values, as the distribution significantly affects the analysis. Second, comparing the distributions of the different utilization metrics helps to highlight the differences between the metric variants. Figure 4 shows a collection of histograms, one for each utilization metric. Each individual chart is a log-scaled histogram that breaks down the components according to their utilization scores for that specific metric. The left column contains the Projects using Component metric, the middle column the Files using Component metric, and the right column the Uses of Component metric. The x-axis shows the utilization metric score for each bin of components on a log scale, while the y-axis shows the fraction of total components falling within that score bin.

These three histograms make it immediately apparent that the distributions of the various utilization metrics are quite different. Before delving into the differences, it is first worth noting the similarities. Every one of the distributions is right skewed, as each histogram had to be presented on a log x-axis. This indicates that no matter the metric, the majority of components show low utilization. This result is not unexpected, and confirms the intuition that some components are very heavily used while most components are hardly used at all. This heavy right skew also means that any scatter plots with the utilization metrics will have the majority of points clustered at low values if a log transformation is not done.

Despite some similarities, the different utilization metrics show quite different distributions. We can see that Projects using Component is the most heavily right skewed of the metrics, with Files Using Component less right skewed, and Uses of Component appearing almost symmetric, after the log transformation. This difference suggests that the number of projects using a component is relatively decoupled from how heavily a component is used by the projects that use it. For example, two components with identical utilization according to Projects using Component may have widely divergent Uses of Component if one component, such as a authentication component, is used only sparingly by projects, while the other component, such as a logging component, is used widely through many files. We can draw this conclusion because, if it were not the case, then the distributions would look identical. Instead, the spread of utilization values becomes wider, with fewer components clustered at extremely low utilization numbers. This suggests that a component’s popularity is not only influenced by how much it does for a project, but also by what it does being important to that project.

Thus, we perform our analysis using all the three utilization metrics. However, due to space constraints we report our results on Projects using Component utilization metric, as our findings are mostly consistent across all the three utilization metrics, without any contradictory observations. We choose Projects Using Component for representation as it best captures the intuition behind component utilization.

RQ1: Is there an inverse correlation between component utilization and its defect density?

We found negligible positive correlation (0.09) between component’s defect density and popularity for the complete set of 2,406 components (row 1, Table III). This is contrary to the common belief that highly used components have lower defect density.

In order to further investigate the relationship, we check if there is any change in this relationship when we only use components which have utilization above a given threshold. Table III shows the correlation between defect density and component utilization for different values of utilization as specified in Column 2 (Component Utilization). Column 1 (No. of Components) represents the number of Maven components which have utilization greater than or equal to the specified threshold. The Pearson correlation\(^3\) between utilization and defect density (computed separately using for HIGH, LOW, and All (HIGH+LOW) priority defects) is shown in Column 3. As shown, whenever a correlation value is significant, it appears to be weak or negligible. Although for components that are used in more than 21,000 projects, we surprisingly see a moderate positive relationship. However, note that even these highly used components bear only weak positive relationship with defect density when computed using only HIGH priority bugs (0.154). Figure 5 showing three scatter plots of defect density and Projects Using Component utilization metric for LOW priority, HIGH priority, and ALL defects (left-right).

Thus, we didn’t find any evidence indicating an inverse relationship between component’s popularity and its defect density.

\(^3\)Statistically significant values (p <0.05) are are marked with (*)
density.

**RQ2: Do highly popular components have better values of software quality metrics than their less popular counterparts?**

Table IV shows the correlation between component utilization and 9 software quality metrics for all the three utilization metrics. As shown, the correlation values range from 0 to 0.25 (for Projects Using Components utilization metric). Manual inspection of the scatter-plots (Figure 6) confirmed that the correlations that fell near zero indeed represented pairings without an identifiable relationship. Note that column 2 (Correlation with Component Size) in Table V contains the results of our analysis. The fact that most of these metrics are highly correlated with component size is a strong indication that component size is serving as a confounding factor.

Out of nine, six metrics show weak but statistically significant relationship with utilization. We analyze the observations for these metrics below:

**Efferent Coupling, Cyclomatic Complexity, Ratio of Derived to Base Classes & Vocabulary Frequency** show correlation coefficient of -0.052, 0.052, 0.117 and 0.041 respectively. Such low values indicating no or negligible relationship of these quality metrics with component utilization.

**Depth of Inheritance** showed correlation coefficient value of 0.218 indicating a weak positive relationship, suggesting that highly used components exhibit comparatively higher depth of inheritance. Since DOI is a regarded as a measure of complexity, a positive correlation means highly used components seem to be more complex than their lesser used counterparts. Similarly, **Afferent Coupling** also showed correlation coefficient value of 0.24 indicating a weak positive relationship.

**Lack of Cohesion & Class Comment Frequency** showed extremely low but statistically insignificant values. Nonetheless, we found that the **Class Comment Frequency** metric shows statistically significant but very low correlation value with other utilization metrics, suggesting no or negligible relationship.

In order to explain the small correlations found above, we considered the possibility that component size is serving as a confounding factor. As such, we performed two additional analyses. First, we computed the Pearson partial correlation coefficient between each software quality metric and component size. The goal was to determine which quality metrics, if any, are correlated with component size. Any metrics that show a strong positive correlation with component size are likely candidates to have their correlation driven by component size.

Column 2 (Correlation with Component Size) in Table V contains the results of our analysis. The fact that most of these metrics are highly correlated with component size is a strong indication that component size is serving as a confounding factor.

The second analysis we performed was to compute the Pearson partial correlation coefficient between each of these quality metrics and the utilization metrics, using size as the controlling variable. A partial correlation coefficient allows a third variable to be introduced into the correlation coefficient calculation, where the confounding effects of that variable are statistically accounted for when computing the correlation coefficient between the primary variables. If component size is in fact driving the correlation between these quality metrics and the utilization metrics, then controlling for component size should reduce the correlations to near 0. Column 3 (Partial Correlation) in Table V contains the partial correlation results, and confirms this belief. For example, as shown previously, the weak positive correlation of component utilization with Depth of Inheritance and Afferent Coupling metrics (0.218 & 0.240) reduces to no correlation (0.081 & -0.018) when controlling for size. Interestingly, this can be explained by the fact that they happen to bear a moderate to strong correlation with size (0.356 & 0.606). Hence, it is very likely that their values are latent driven by component size.

These two analyses allow us to conclude that the relationship we are seeing between some of the quality metrics and the utilization metrics is largely driven by the confounding effect of component size. Also note, that re-evaluating RQ1 by computing partial correlations to control for size, further reduces the strength of correlation. Thus, all the correlation values shown in RQ1 and RQ2 are further reduced when controlling for size providing further support that we did not

---

**TABLE III**

**Pearson Correlation between Projects using Component utilization metric and Defect Density computed using LOW Priority, HIGH Priority, and ALL the Defects.**

<table>
<thead>
<tr>
<th>No. of Components</th>
<th>Component Usage (&gt;0)</th>
<th>Correlation Coefficient (utilization &amp; defect density)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>All Defects</td>
<td>High Priority Defects</td>
</tr>
<tr>
<td>2406</td>
<td>1</td>
<td>0.09*</td>
</tr>
<tr>
<td>855</td>
<td>20</td>
<td>0.063*</td>
</tr>
<tr>
<td>554</td>
<td>100</td>
<td>0.106*</td>
</tr>
<tr>
<td>353</td>
<td>500</td>
<td>0.087*</td>
</tr>
<tr>
<td>288</td>
<td>1,000</td>
<td>0.171*</td>
</tr>
<tr>
<td>212</td>
<td>2,000</td>
<td>0.128*</td>
</tr>
<tr>
<td>171</td>
<td>3,500</td>
<td>0.123*</td>
</tr>
<tr>
<td>144</td>
<td>5,000</td>
<td>-0.102</td>
</tr>
<tr>
<td>108</td>
<td>8,000</td>
<td>-0.051</td>
</tr>
<tr>
<td>96</td>
<td>10,000</td>
<td>-0.013</td>
</tr>
<tr>
<td>75</td>
<td>12,000</td>
<td>-0.150</td>
</tr>
<tr>
<td>54</td>
<td>15,000</td>
<td>0.140</td>
</tr>
<tr>
<td>35</td>
<td>18,000</td>
<td>0.195</td>
</tr>
<tr>
<td>26</td>
<td>21,000</td>
<td>0.351*</td>
</tr>
</tbody>
</table>

---

**TABLE IV**

**Pearson Correlation between utilization metrics and software quality metrics.**

<table>
<thead>
<tr>
<th>Metrics</th>
<th>Sign</th>
<th>Projects Using Component</th>
<th>Files Using Component</th>
<th>Uses of Component</th>
</tr>
</thead>
<tbody>
<tr>
<td>Efferent Coupling</td>
<td>-</td>
<td>-0.052*</td>
<td>-0.003</td>
<td>0.025</td>
</tr>
<tr>
<td>Afferent Coupling</td>
<td>+</td>
<td>0.240*</td>
<td>0.321*</td>
<td>0.388*</td>
</tr>
<tr>
<td>Lack of Cohesion, Field</td>
<td>-</td>
<td>-0.003</td>
<td>0.001</td>
<td>0.012</td>
</tr>
<tr>
<td>Depth of Inheritance</td>
<td>-</td>
<td>0.218*</td>
<td>0.233*</td>
<td>0.234*</td>
</tr>
<tr>
<td>Ratio of Derived to Base Interface</td>
<td>-</td>
<td>0.117*</td>
<td>0.130*</td>
<td>0.131*</td>
</tr>
<tr>
<td>Weighted method per class</td>
<td>-</td>
<td>0.031</td>
<td>0.04*</td>
<td>0.029</td>
</tr>
<tr>
<td>Vocabulary Frequency</td>
<td>+</td>
<td>0.041*</td>
<td>0.056*</td>
<td>0.097*</td>
</tr>
<tr>
<td>Class Comment Frequency</td>
<td>+</td>
<td>-0.015</td>
<td>0.098*</td>
<td>-0.085*</td>
</tr>
<tr>
<td>Cyclomatic Complexity</td>
<td>-</td>
<td>0.007*</td>
<td>0.056*</td>
<td>0.042*</td>
</tr>
<tr>
<td>HIGH priority Bugs</td>
<td>-</td>
<td>0.224*</td>
<td>0.240*</td>
<td>0.263*</td>
</tr>
<tr>
<td>LOW priority Bugs</td>
<td>-</td>
<td>0.177*</td>
<td>0.166*</td>
<td>0.185*</td>
</tr>
<tr>
<td>All Bugs</td>
<td>-</td>
<td>0.167*</td>
<td>0.155*</td>
<td>0.173*</td>
</tr>
</tbody>
</table>
find any evidence supporting the belief that highly utilized components have higher quality compared to their lesser used counterparts.

TABLE V

<table>
<thead>
<tr>
<th>Metrics</th>
<th>Correlation with Component Size</th>
<th>Partial Correlation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Efferent Coupling</td>
<td>0.102*</td>
<td>-0.105</td>
</tr>
<tr>
<td>Afferent Coupling</td>
<td>0.606*</td>
<td>-0.018</td>
</tr>
<tr>
<td>Lack of Cohesion, Field</td>
<td>0.066*</td>
<td>-0.033</td>
</tr>
<tr>
<td>Depth of Inheritance</td>
<td>0.356*</td>
<td>0.081</td>
</tr>
<tr>
<td>Ratio of Derived to Base Interface</td>
<td>0.272*</td>
<td>0.003</td>
</tr>
<tr>
<td>Weighted method per class</td>
<td>0.027</td>
<td>0.021</td>
</tr>
<tr>
<td>Vocabulary Frequency</td>
<td>0.416*</td>
<td>-0.049</td>
</tr>
<tr>
<td>Class Comment Frequency</td>
<td>0.040*</td>
<td>-0.021</td>
</tr>
<tr>
<td>Cyclomatic Complexity</td>
<td>0.088*</td>
<td>0.016</td>
</tr>
<tr>
<td>HIGH priority Bugs</td>
<td>0.389*</td>
<td>0.074</td>
</tr>
<tr>
<td>LOW priority Bugs</td>
<td>0.389*</td>
<td>0.017</td>
</tr>
<tr>
<td>All Bugs</td>
<td>0.359*</td>
<td>0.02</td>
</tr>
</tbody>
</table>

V. RELATED WORK

A number of empirical studies have looked at component utilization within various operating system ecosystems, such as Linux [15], [16] or Unix [17], whose structured nature makes them significantly more amenable to analysis. Researchers, such as Lungu et al. with their Small Project Observatory [29], have also explored the visualization of component dependencies, though on a relatively modest scale. Lammi et al. used a method similar to ours, analysing a collection of 6,286 projects drawn from SourceForge [30]. However, they only analyzed projects that could be fully compiled, and only assessed a limited set of components against this repository.

With regards to software quality metrics, many of the initial studies evaluating their effectiveness focused on a very limited number of systems [9], [11], [12]. This was likely due to a combination of the limited availability of software to examine and computational concerns. Thanks to the advent of open source software and improving hardware technology, more recent studies have scaled up the set of projects used for the validation [10], [31], [32]. However, the number of projects studied is still usually in the dozens, vastly less than the number of projects examined in this paper. A group of researchers has attempted to enable the larger-scale analysis of software quality metrics through the introduction of Alitheia, a platform for monitoring software quality [33].

Researchers that have studied utilization at a type-based granularity have primarily been interested in building code search engines, and using the resulting utilization information to aid in ranking. This approach was seen in early versions of the Sourcerer Code Search Engine [34], as well as Spars-J [35], Merobase [36] and the more recent Portfolio [37]. These different systems generally assess utilization by attempting to match type names used in one file against definitions in another. The specific approaches range from simple text matching to automated dependency resolution [38] to partial program analysis [39].

Studies of limited sets of components have mainly focused on identifying API hotspots [40], effectively assessing what portions of the components are heavily used, rather than which components are used more heavily. The study of API usage is similar to that of component utilization, yet more limited in scope. The goal of component utilization is to globally determine the usage of all components. In API usage, however, specific components are picked and their usage explored. API usage therefore can lead to an understanding of how specific components are used, but not to a global picture of component
utilization. API usage has primarily been studied using two methods. Researchers either evaluate a component’s usage within a collection of projects [41], [30], or by using leverage of existing internet or code search engines [42], [40], [43]. Component utilization is computed using the first method, a collection of projects, with the requirement that the collection be sufficiently large.

While small-scale studies of software quality make sense and can provide valuable results, it is difficult to assess component popularity without the use of a large dataset. Previous studies of utilization have had two primary limitations. Either they focused on an extremely limited set of manually identified components, or they assessed utilization at a type-based granularity rather than component-based. We overcame these difficulties using Maven components. Thus, while there have been a large number of empirical studies on software quality metrics and component selection separately on limited datasets, we are not aware of previous work that has performed a large-scale empirical analysis of their relationship.

VI. THREATS TO VALIDITY

This is a large-scale empirical study based on software components in the real-world. As is usually the case for studies of this kind, many heuristics and approximations needed to be made. One of the most problematic approximations may be the use of aggregated components.

In real-world software, versioning poses many challenges to these kinds of studies. Components evolve, and the projects that use them end up using many versions of them. The identification of the exact version of a component that a project uses is a difficult problem to solve. Even if we had solved that problem, we would still have to face the fact that the 55,000+ “consumer” projects all use different versions of the Maven components. Had we restricted the consumer projects to those that use the exact same versions of Maven components, our dataset would be insignificantly small, and the study would not be valid. As such, we needed to have some estimate of a component’s popularity and quality independent of specific versions that the consumer projects use.

We addressed this issue by using the concept of aggregated components explained in Section II-A, which is the collection of all versions of a project in Maven. Our metrics include the entire collection of versions, and, when needed, we normalize them by dividing by the number of versions or by the total lines of code.

This method is not ideal, as projects sometimes go through significant changes from version to version – in size, defect density, etc. The method used gives a rough estimate of the projects’ properties independent of the projects’ evolution. Better methods for estimating the essence of components over time would be desirable.

Another threat is that 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 components, and found no evidence of systematic use of FindBugs tool. We did this using by going through the developer list, and examining the history of warning counts of these projects. In addition, we found no evidence in the email archives of any these projects suggesting a systematic adoption FindBugs tools. These observations provide some mitigation to this particular threat to our findings. Moreover, even if there are few projects that have adopted FindBugs as part of their development, we believe the study sample is large enough to eliminate bias of this kind.

VII. CONCLUSIONS

This paper presented an empirical analysis of the relationship between the utilization of open source components and their quality. In this study, we determined the exposure (popularity) of 2,406 Maven components by calculating their usage across 55,191 open source Java projects. As a proxy of code quality, we calculated (i) defect density using the set of bug patterns reported by FindBugs; and (ii) 9 popular SQO-OSS quality metrics. We then looked for correlations between popularity & quality as calculated above.

The results in this study do not support one of the most popular beliefs about open source software. While many eyes may very well increase code quality in some projects, statistically speaking that effect doesn’t seem to exist within one very popular component repository, Maven. Another conclusion of this study is that the utilization of open source components is driven by factors other than their quality, a result with implications for software engineering research.

Results for the Java ecosystem may not generalize to the open source projects as a whole, as other programming languages may force unique constraints on their users which alter the impact of quality on utilization. The conclusions in this study should not be extrapolated to other language ecosystems.

Reproducibility. We welcome other researchers to reproduce and even replicate the study on other software systems. For this, we have made available all the necessary artifacts including project sources and raw data; detailed steps including artifacts to process data; and analysis procedure to reproduce the statistical results to verify the claims. All artifacts will be made available at the following URL: http://mondego.ics.uci.edu/projects/mavenpopandqual.

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

REFERENCES


