A novel approach to static code analysis and software quality assessment

Brian S. Dillon

Abstract— A number of static code analysis tools have been proposed for the automatic defect detection and quality assessment. Many static code analysis tools use a similar approach to detecting defects employing the Visitor Pattern to traverse code and search for patterns of interest. This approach is different from the approach used in manual code reviews. This paper proposes a novel set of quality assessment algorithms designed to replicate the action of human reviewers and proposes a novel approach to determine software quality from metadata inherent in the code. The paper uses the metadata approach to compare the novel algorithms with legacy static analysis tools commonly found in the literature and calculates correlation among the various tools. The paper concludes that the novel algorithms have no correlation with legacy tools, that some legacy tools perform no better than random quality assignment, and that the novel algorithm performance is a statistically significant improvement over some legacy tools.

Index Terms— Quality analysis and evaluation, Software Quality Analysis, Quality Concepts, Code inspections and walkthroughs, Review and evaluation.

1 Introduction

TRAJCHEVSKA [21] made it clear that there are significant differences between new start, maintenance, and legacy software development. Speaking from personal experience, when working to restart legacy code, "[y]ou spend more time reading than writing code." This is a function of the constrained environment where any decision must take into account the considerable momentum of the existing design.

When working a new start, the developer is relatively unconstrained, adding features in a dynamic environment. Quoting Trajchevska, "I really didn't have time to think about the code structure—at least I didn't think I had enough time to do it." All good things come to an end, however, and within short order that same code entered 'maintenance mode'. It became necessary for the developer to spend an extraordinary amount of time looking through her own code—written just two years before—just to find portions of code relevant to a desired change. That new start code created its own constraints that inevitably affected future changes. New starts always become legacy code.

These three stages of development represent a continuum from high-knowledge and low-constraints to low-knowledge and high constraints. New starts have all the freedom to leverage any number of tools, libraries, and design patterns to enable more reuse and less recoding. Every part of the program is new and intimately known. As the program enters the maintenance stage, past design decisions increasingly constrain future decisions, resisting change and encouraging continuity. As the scope of the program continues to grow, the developer becomes less intimately familiar with the majority of the program. In a legacy restart, constraints abound while knowledge is non-existent. In the worst cases, the code may not even compile and any documentation is superficial at best.

Trajchevska's personal experience is not unusual. Van Gurp and Bosch [23] discussed the same design momentum problem, stating that every new decision either has to conform to past decisions or else "vaporize" those decisions at the cost of considerable rework. Parnas [15] indicates that all maintenance actions are constrained by knowledge and that uninformed developers are a danger, performing "ignorant surgery" on the program. Eick [6] identifies organizational churn as a contributing factor to bad code as it "degrade[s] the knowledge base and can also increase the likelihood of inexperienced developers changing the code." Lehman's fifth law [12] also speaks to this phenomenon. In his words, the "rate and quality of progress and other parameters are influenced, even limited, by the rate of acquisition of the necessary information by the participants collectively and individually."

Quality of code is inextricably connected to the knowledge of the developers—their familiarity with the code, its design, and the effect of every change. When developers lack this knowledge base, quality will decline and maintenance efforts will grind to a halt. Moreover, the constraints of past decisions—especially poor decisions—will increase the cost of each change. In a 2015 talk, Pizka [16] spoke to this phenomenon in what he called Return on Investment (ROI) death. In summary, ROI death is when quality control issues increasingly strangle the development budget until all innovation and new features cease. Maintaining the program is no longer financially feasible because there is no return (i.e. added value) on the investment.

• *B.S. Dillon is a lead scientist at the Naval Surface Warfare Center in Dahlgren, Virginia and a PhD student at Virginia Polytechnic University and State School in Blacksburg, Virginia. E-mail: brian.s.dillon@navy.mil.*

How many programs have been cancelled and rewritten because continued maintenance is no longer financially feasible? Spolsky is on record [19] against this practice, citing the history of otherwise successful software (e.g Netscape) that suffered after rewriting. Nevertheless, business decisions are made every day to rebuild from the ground up. Two typical strategies are to (a) completely scrap inflexible software or (b) rescue only the most critical functions. These business decisions recognize (on some level) the realities of ROI death and the reality that developers cannot be familiar with all of their own code.

In 2011, the author was given the task of restarting a legacy program developed by a subcontractor. As the program was still in use, management wanted to extend its life for a few years until a replacement could be developed. With only one developer, however, restarting the 300K lines of code (LOC) program was just not possible. It took a team of five people working for one year just to get the code to compile and deploy again. Little knowledge of the code, aggravated by the large size of the project, made it a daunting and painful process to get the code running. To make intelligent and reasoned improvements to its design and function would require even greater effort.

This unwieldy task led to a number of questions. The crux of the problem is the amount of labor required to evaluate any sufficiently complex design. Manual code reviews are effective means of achieving these goals but are extortionately expensive at this scale. Are manual code reviews the only means to identify these code features? Is there an automated alternative to code reviews with the same efficacy? Similarly, there are already a number of static code analyzers. What is the performance of static code analysis when compared with manual code reviews by a human? Is there a way to execute an exhaustive human-like review in a fully automated way? The author set out to develop such automation and to compare those results with outputs from existent static analysis tools.

1.1 Related Work

Wagner conducted a study [24] comparing the effectiveness of static analysis tools, testing, and code reviews in defect detection. The study defined five defect categories:

1. Critical defects leading to a crash.
2. Critical defects leading in incorrect results.
3. Non-critical defects improperly handled.
4. Poor practices adversely affecting the program execution e.g. performance.
5. Poor practices adversely affecting the readability and maintainability of the program.

Wagner's study found that static analysis tools were primarily able to detect defects in categories 4 and 5. Furthermore, the tools generated a substantial number of false positives among the findings—66% false positives on average. Wagner said that the quality of findings were highly dependent on "how good and generic the [bug detection] patterns can be." Far from reducing human effort, with the high number of false positives "human effort could be even higher" when using static analysis tools.

Wagner found that software testing detected defects in categories 1, 2, and 3 and human reviews detected defects in all five areas. While "tools are better regarding the bug patterns they are programmed for" human led review "is able to detect far more defect types." On the other hand, software testing was complementary to tools as "dynamic test techniques find completely different defects" from static analysis techniques.

Wagner's concluded that a combination of techniques is necessary to achieve the best results. In theory, tools can identify areas of high interest for subsequent, human-led review. This targeted approach would benefit from fewer false positives possibly assisted by annotations in the code that would "allow the tool to understand the code to a certain extent and therefore permit some checks of the logic." In other words, better tools might get better results, but human-led review is essential.

Another study [18] by Rutar compared the abilities of various static analysis tools: ESC, FindBugs, JLint, and PMD. This study identified 12 unique defect types from these tools by analyzing 5 software projects. It also considered the potential for a meta-tool that would combine all findings. Defect density varied highly between tools and projects from 0.1% to 23.2% with a cumulative defect density of 26.3% across all projects and tools.

The study concluded that combining defect reports from various tools into a meta-tool might have additional affordances. Code reviewers would take particular interest in any class with an elevated number of unique or total reports. Individual reports would prune or elevate one another based on complementary results. Again, the paper suggested that annotations could reduce false positives and the overall number of defects.

Rutar, however, noted that actual false positive or false negative rates were never established by the study. "Doing so would be extremely difficult, given the large number of warnings from the tools and the fact that we ourselves did not write the benchmark programs in our study." In other words, the burden of work alluded to by Wagner was a deterent to Rutar. The sheer number of findings precludes exhaustive evaluation. Unfamiliarity with the code also elevated the level of effort required to perform that evaluation.

Hovemeyer conducted another examination of the tool FindBugs [11]. That study reviewed findings from FindBugs across six programs with a highly variable defect density (2.2%-78.6%) and an equally variable false positive rate (13.6%-45.5%). Hovemeyer noted that the triage of tool findings "is a manual, timeconsuming and subjective process." Hovenmeyer also indicated that tools can find defects that would otherwise go undetected. As with Wagner, Hovenmeyer concluded that static analysis tools best serve to bring potential defects to the developers' attention.

A different perspective comes from Ayewah [1]. This study evaluated not the precision of the findings, but the impact. Ayewah defined categories for impossible defects that "could not be exhibited by the code" and trivial defects that "might happen but would have minimal adverse impact." Suppose a buffer overflow occurs at x>100, but

the control structure only executes that branch if x<50. Because the error cannot occur, there can be no impact on function. This method of evaluation brings into questions the validity of false and true positive measures in all other studies.

Ayewah's studied 1127 different reports, finding that many could have only a trivial (11%) or no impact on function. Of the remaining defects, 46% had been fixed by developers, leaving 26% "open" i.e. considered significant, but not yet fixed. If these findings are representative of all code, then static analysis tools may turn out a mixture of false positives, true positives, and true defects that simply don't matter. Is there a null pointer? Possibly, but we already check for null pointers, so it is a moot issue.

Finally, let us consider Kim [14] where the meta-tool concept was employed to good effect. Kim used three static analysis tools, training the meta-tool to give weights for each defect detection algorithm based on historical precision data. The paper also compared the top 20 reports ranked by historical warning precision (HWP-20) and tool warning precision (TWP-20) which overlapped by 50%.

Although Kim achieved good results with a highly tuned weighting system, the weighting system itself reveals interesting details about the warnings in the study. Of the HWP-20, only two reports had weights above 1.0 (2.8 and 1.1). The rest had weights under 0.5 with an average weight of only 0.428 for the HWP-20. Among the TWP-20, two weights were zero and the average weight was far lower at 0.162. What do these low weights reveal about the value of the warnings?

Suppose that, as a developer, you receive the reports from these tools. Can you act on that information? According to these weights, 531 datapoints from the TWP-20 render only 6.677 (1.3%) units of useful data. The most verbosely reported findings—responsible for 449 of the reports and none with a weight above 0.01— were responsible for this poor showing. Among the less verbose HWP-20, the 23 findings render more useful data—9.426 (41.0%) units of information. In short, the more frequently reported findings are providing very little actionable data and the more valuable results are hidden wheat among the many tares. Kim's success with a meta-tool is achieved mainly by dampening the noise in the signal.

Other readings show the same pattern of behavior for static analysis tools. Note that the timbre of most papers in this area is hopeful and encouraging in spite of a lackluster performance by the tools. Clearly, the community hungers for a useful set of tools that statically evaluate source coded and provide actionable data. Additional works in this area from the most recent years are:

- Gupta [9] used Coverity® because it was a "third generation tool." The study found that only 8% of Coverity findings were resolved compared with 95% of crash reports.
- Tomassi [20] revised a previous study where ErrorProne and SpotBugs missed 95% of known defects. In the new study, favoring each tool's best area of performance, the tools still failed to detect 60%-70% of all defects.
- Li [13] found an average defect density of 0.59 (open source) and 0.72 (closed source) using Coverity across a number of projects—supposedly a new error on every other line.

1.2 Static Analysis: Visitor Pattern vs Crawler

Based on the body of available work, current static code analysis tools are not terribly effective or precise, providing an overly verbose number of low-value, suspect reports. One reason for the ineffectiveness may come from a reliance on the Visitor Pattern [7]. The framework of the typical static analysis tool exists to step through the code and expose incremental portions of source code to various defect detection algorithms. Those algorithms are Visitors only and have no ability to review outside of the limited area that is offered for examination. Defect detection is limited to the scope of a single code block or even one line. The Visitor pattern inhibits the development of an algorithm that spans multiple method calls and classes so that findings are limited in scope and importance as noted by Wagner [24].

Human code analysis uses a completely different process. Once a human sees something suspicious or finds a specific anti-pattern, he can conduct a series of analyses branching out from that point. How is this method used? How is it called? Where? What is the state of the program at the time? As described in Ayewah [1], a static analyzer is incapable of more than simple pattern matching while a human can deteremine whether or not the pattern is of any importance.

The ideal static analysis tool would work more like a human reviewer. Such a tool may start from a matching pattern, but branch out from that position to gather further evidence to support or refute the initial suspicion. The process is very like web crawler algorithms where the crawler gathers data and charts its own course across various sites. This paper evaluates a novel set of algorithms, described in [3] that attempt to replicate the action of human code reviewers [4] using the crawler model. These algorithms explore a graph that represents control-flow, data-flow, and structural relationships within the code. It is hypothesized that algorithms built on the crawler model will be more effective in discovering design-level defects within source code because they replicate, in part, the same evaluation processes used by human code reviewers.

To investigate the crawler approach a prototype framework was developed using ANother Tool for Language Recognition (ANTLR) and a Java® grammar to build the required graph. The resulting graph was subjected to analysis by several crawler algorithms, each of which annotated the findings for every field, function, and file. The framework is not generally applicable as the grammar exclusively supports Java. The graph-based crawler algorithms are more generalizable, as they relate directly to principles of object-oriented design and logical analysis. At the time of writing, the framework has not been extended to work with other languages.

2 METHODOLOGY

2.1 Sampling Method

This study compares the ability of the novel set of algorithms, collectively known as Software Quality Metrics (SQM), with those of several well-known, "legacy" static analysis tools. Each of the tools analyzed several samples of Java source code. The samples were taken from thirteen different software projects from open source, commercial, and military domains. Files from each project were sorted and divided into 10 buckets of increasing size as measured by LOC. Five files were selected at random from each of the 10 buckets, creating a sample of 50 code files from each project. This selection method ensures that each sample was random and represented code from across the whole project without bias for size. The final sample included 650 code files.

2.2 Aggregating Static Analysis Results

Various methods have been proposed to form aggregate values from independent defect detection algorithms. Zhang [25] conducted a study of these methods. Zhang concluded that no single method is free from error and opted for a meta-value encompassing input from each method. With no clear rationale to select one aggregation method over another, other factors may become more important. Various reports from legacy tools and SQM apply at the class level. Instead of devising a mechanism to divide class-level values into smaller components, it is convenient to create a direct summation aggregate at the class level.

Originally, PMD was selected as the legacy tool for comparison with SQM. Additional tools were added to expand the study beyond a simple head-to-head comparison. SonarQube® and CheckStyle were included because they were components in a metatool available to the author. Note that all three legacy tools show up in other studies within the literature. SpotBugs or FindBugs were considered but ultimately rejected because few of the 13 sample projects would compile into the Java Archive (JAR) files necessary for those tools.

Each tool produced a unique distribution of scores over the entire sample, based on verbosity of the various algorithms and the defects they detected. In spite of the unique distribution ranges, the distribution curves are fairly uniform in appearance. The distribution for PMD, shown in Figure 1, is typical—a gamma distribution with a mean close to zero and a long tailoff. The majority of of code files have very few defect reports, but some individual files had an extremely high number of reports. In spite of this distribution, when the PMD findings are ranked they show a strong positive correlation with LOC, as shown in Figure 2. While this correlation is not unexpected, it is remarkable that the results of such a popular open-source tool are closely predicted by LOC alone.

SQM is different from the legacy tools in one important aspect: it includes reports of both good and bad coding practices. This difference produces the two independent distributions of good (white) and bad (black) values visible in Figure 3. Inevitably, some code files have a mixture of good and bad findings and these are considered as a ratio score (#Bad/#Good). In a minority (7%) of these cases, the ratio is overwhelmingly bad but roughly 25% of all files fall within the interval (0,1] in gray. It is debatable how to score the gray section. The analysis section will discuss that disposition at greater length.

The full distribution of SQM values is on the interval [-53, 11], but 99% of that range is shown in Figure 3. Approximately 25% (striped) of all files have no SQM findings. Those files were excluded from further analysis.

As noted previously, PMD has a 78% correlation with LOC. CheckStyle has a similar correlation at 75% and SonarQube has a weaker correlation at just 51%. Because of this evidence, LOC is included in the data and analysis of this study as if it were a separate tool. It is instructive to

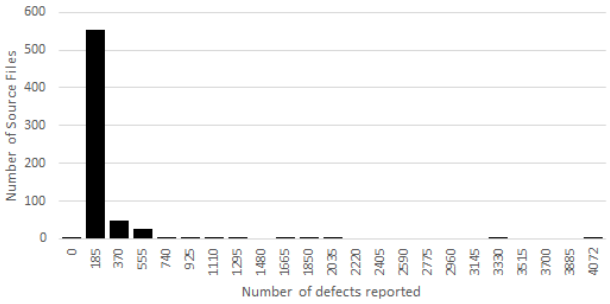


Fig. 1 Distribution of PMD defect reports per source file.

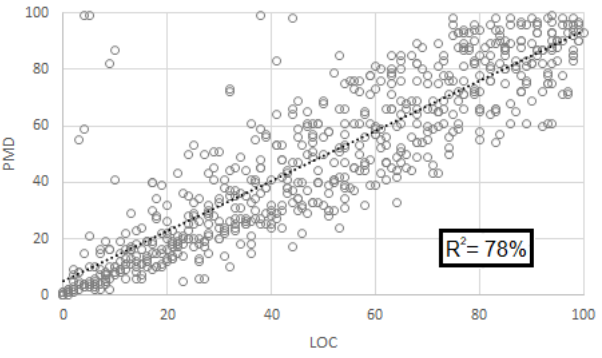


Fig. 2 Ranked PMD findings correlated with Lines of Code.

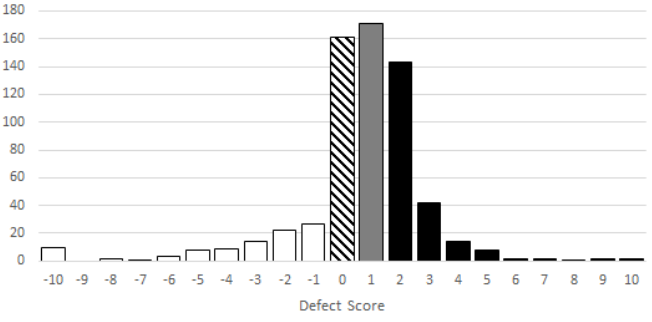
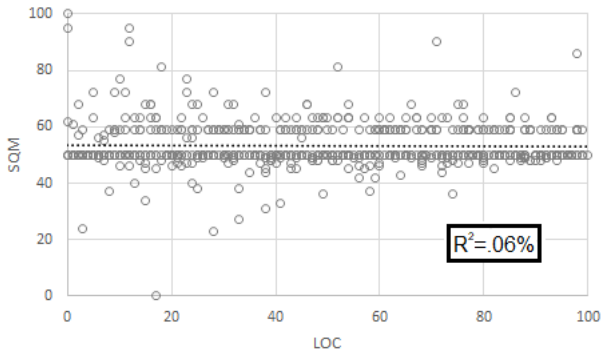
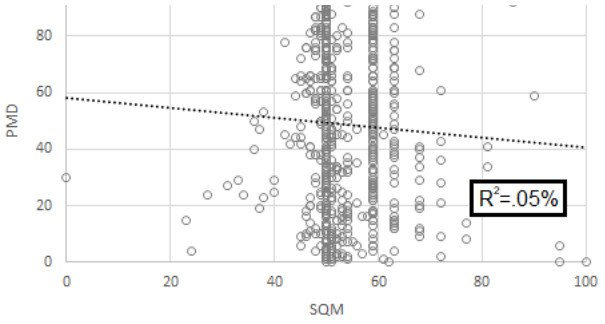


Fig. 3 Distribution of SQM scores per source file.



rank the SQM scores in the same manner. SQM has no cor-

Fig. 4 Correlation of ranked SQM scores with LOC.



relation with LOC, showing an evenly distributed set of

Fig. 6 Correlation of SQM and PMD scores.

scores across all LOC values and a negligible negative correlation (R^2=0.06%). This is remarkable evidence that SQM is measuring some value that is distinct from LOC—a claim the other tools cannot make.

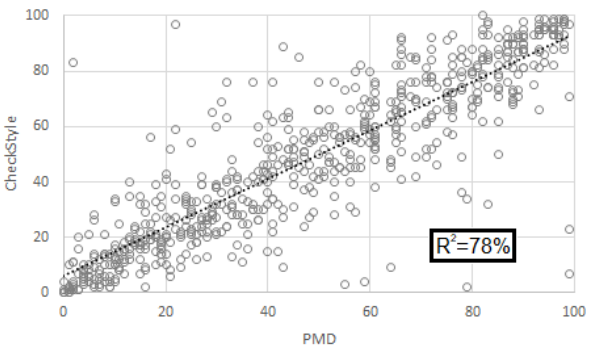
It is also instructive to show that PMD and CheckStyle have a strong (R^2=78%) agreement with each other while SonarQube has a weaker agreement (R^2=51%-55%) with either of the other legacy tools. SQM uniformly shows a negligible negative correlation with any of these tools (R^2=0.05%-0.02%). Again, this is remarkable evidence that, whatever the legacy tools are measuring, SQM is measuring something far different.

Figures 4 through 7 show exemplary correlations of ranked tool scores and LOC.

2.3 Objective Quality Measures

The proposed study also requires a standard to judge the truth of the classification. Chidamber & Kremerer [2], among many other, have proposed numeric metrics that may be easily calculated by a brief examination of the source code. Pizka [16] also warns that numeric metrics are "easy to measure instead of important to know". The strongest claim offered by proponents of numeric metrics is generally that files with unusual numbers "are candidates for inspection and/or revision" [17]. In this study, the objections to numeric metrics are practical:

- If any of the four tools correlates with a numeric metric then that tool has, at best, found a more expensive way to calculate the same value.



- If the tool does not correlate with a numeric metric there is no firm measure of ground truth to indicate which is

Fig. 5 Correlation of PMD and CheckStyle scores.

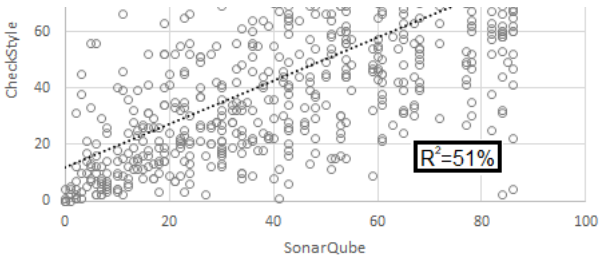


Fig. 7 Correlation of CheckStyle and SonarQube scores.

more successful.

It is popular wisdom that "[t]he only valid measurement of code quality [is] WTFs/minute" [10]. In other words, expert code reviews would be the best source of truth data. Nevertheless, early experimentation with code reviews showed a high correlation with LOC. Reviewers tended to ignore obvious faults in code, relying—perhaps subconsciously—on the apparent size of the code file. There are several reasons why manual code reviews are less than ideal as a source of truth:

- Excessive amounts of human time required;
- Cognitive shortcuts taken to reduce that time;
- Variation in skill levels among reviewers;
- Subjective interpretation of 'quality';
- Limited familiarity with the specific code base;
- Limited review boundaries, e.g. one class or method in a vacuum.

In order to arrive at a concrete, objective measure of quality from developers, it is necessary to review the artifacts created during development. Consider that every identifier and comment in the code represents information derived from the developers. Those metadata are different from the function and correctness of the code itself; they represent a well of information about the state of mind of the developer during the development process. Therefore, comments and other metadata in the code can serve as an analog measure of the mindset of the developers and their interpretation of the quality of the code.

The metadata of interest falls into three broad categories with examples in figures 8 through 10 taken from sample code. The first category is 'eccentric coding practices'. Eccentric coding practices are not functional defects, style guide violations, or simply unusual behavior. Each of these examples is demonstrably the result of poor quality control:

a. This method and its header comment are correct. The pair obviously go together.
b. This method maintains a header comment apparently derived from copying the method in example **a**—a discrepancy that will appear in the JavaDocs.
c. It is common practice to identify methods that return a Boolean value with "is" verb phrases. In this case, whether the field name was correct or not, compounding the verb phrase is just confusing.
d. This method is a tautology, always allowing the operation. A child class could replace a protected method like this. An investigation of the many child classes, however, found no cases where the default behavior was changed.
e. As with method **c**, the double verb phrase is an obvious error. The double 'to' is just a bonus.

The blame for eccentric coding practices like these does not lie exclusively with the developer. A code review by a peer should catch these obvious problems. Because they were not, either the code was not reviewed or the developers were not vigilant enough. A lack of thoroughness may also stem from time constraints on the reviewers. In such a constrained environment, quality may suffer.

The second category consists of a *cri de coeur* or a heartfelt, emotional outcry. Statements like those in Figure 9 usually stand alone as a demonstration of developer's frustration and dissatisfaction. While a human reviewer may provide live criticism for a section of code, these statements are fossilized evidence that a developer came this way and was highly displeased. These statements almost literally fulfill the fabled requirement for WTFs-per-minute:

f. There is no further explanation for this statement in the code. Presumably, this code would displease Jacob in some way and the writer knew it.
g. In general, `TODO` tags are not a sign of defects, but an indication for future improvement. As such, they are not a cause for alarm. This tag, however, is less than constructive and indicates the frustration of the developer over the situation.
h. Regular expressions are notoriously tricky to work with and small 'improvements' can cause strife among teammates. Apparently, this developer was more aggressive in staking out her territory.
i. This developer was so overwrought that he felt the need to coin new words for this level of badness.
j. As with **g**, this tag indicates only the frustration of the developer with inadequate code.
k. This comment stands alone. It is unclear what didn't work, but the developer is obviously frustrated.
l. This statement indicates a rather pernicious bug exists. That the developer identifies the fix as a "hack" indicates the solution is only just better than the bug itself.
m. This developer felt the need to personally attack another for poor design work. The attack is even autographed for posterity.
n. This developer was having a tough time aligning GUI components. The level of frustration expresses itself in an epithet for that component.

```
/**
 * @return Returns the isSavePasswordEnabled.
 */
a public final boolean isSavePasswordEnabled() {
    return isSavePasswordEnabled;
  }

  /**
   * @return Returns the isSavePasswordEnabled.
   */
b public final boolean isAutoLoginChecked() {
    return autologinCheckbox.isSelected();
  }

c public boolean isIsCrossDomain() {
    return this.isCrossDomain;
  }

d public protected boolean allowToolbarAutoHide() {
    return true;
  }

e public boolean isWasAddedToToMainApplicationWindow() {
    return wasAddedToToMainApplicationWindow
  }
```

Fig. 8 Examples of excentric coding practices.

```
f // Jacob don't kill me...

g //TODO HACK ALERT!!!!!!!!!!!!!!!!

h //think twice before changing the regex

i // this is badwrong, how did we get it in the first place?

j // TODO: kind of a hacky way to minimize to dashboard

k //this didn't work

l // (hack to remove things from the database so
  // they don't appear twice in the gui)

m // AN: Thank you BN for this stupid design....

n // RR: the problem child...
  // how do I line this up the way I want it???
```

Fig. 9 Examples of a cri de Coeur.

These examples show that a *cri de coeur* comes in various flavors. Comments like these are unprofessional and code-cleaning tools routinely strip them out. Even so, they show up in some sections of code and, more often, in commit descriptions. Critical examination of the code should find these comments. That they were not found is further proof that the code in question was not reviewed and has significant quality concerns.

The final category consists of evidence that the developers are aware of low quality indications but they have simply given up. This evidence takes various forms like delaying action, acknowledging but doing nothing about a problem, and derisive or satirical statements. These metadata indicate that the developers are aware of quality control issues, but they have accepted this state as the *status quo*.

o. This is part of a lengthy comment indicating the solution was sub-optimal—a hack—but the developer would eventually fix the problem. This concern should be added to the backlog for future development, not left as a comment in the code. Note the date. At the time of discovery, the fix was already years out of date.

p. This comment reveals a bug in the software, at least hindering the user experience and probably creating an unstable system. Nevertheless, the developer comments on the ridiculous workflow rather than resolving the bug. Grammar and spelling errors are gratuitous.

q. This comment indicates that the developer, perhaps relying on some ineffable impression born of experience, suspects the method is no longer relevant. Still, the developer takes no action, leaving any investigation to the next person.

r. Silently quashing an exception is a bad practice. Not only does the developer acknowledge the impropriety of his actions, but he also expresses the helplessness he feels with respect to the code.

s. This unusual statement concerns network feedback that caused outgoing messages to redirect a copy to the emitting system. The vulgarity of the statement is arguably an eccentric coding practice, but the author appears disheartened by the inability to prevent the feedback. He has given up preventing it and settles for ignoring it.

t. This logging statement exists in a window closing event handler. Probably, the developer wrote the statement to superficially meet some coding standard, but it is obviously inadequate. The developer felt that no useful logging statement was warranted, so a useless one was substituted.

u. This method, without comment or description other than the signature is exactly what it says. To all developers who eventually look on these HEX codes, there is no way to understand what the sequence means or why. That a developer named the method in this way indicates apathy in correcting the situation.

```
o  // I hope to correct this when we revamp the jobs
   // stuff altogether. JB 2015-01-30

p  // This class only exists becasue the regular right
   // mouse menu handling does not work for the buttons
   // that a DataScalePanel consists of. Quite a bit
   // comlicated just to bring up a right mouse menu, isn't it.

q  // TODO - maybe delete this method?

r  catch (BadLocationException ex)
   {
     //Silently ignore, what could we do anyway?
   }

s  //sniffing our own farts here, ignore this data

t  System.out.println("Blah");

u  private void doBlackMagic() throws IOException {
     vendorIn(0x8484, 0, 1);
     vendorIn(0x1308, 0, null);
     vendorIn(0x8484, 0, 1);
     vendorIn(0x8383, 0, 1);
   }
```

Fig. 10 Examples of giving up.

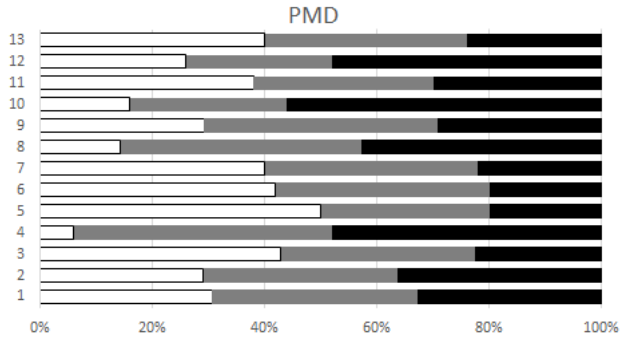


Fig. 11 Variability of 33% buffer range (grey) over 13 projects.

When developers give up on a competent level of code quality, it says something dire about the code itself. Lehman's laws [12] require constant effort to preserve code quality and avoid a downward spiral. After exerting effort to improve code quality, it is disheartening to see the code deteriorate further. Put in Pizka's terms [16], the majority of developers' resources are spent fighting low quality. These metadata indicate the quality of the code is low and has been for some time.

2.4 Statistical Evaluation

Aggregate results from each of the legacy tools was partitioned at the median value. This partition created a binary classification. Class files that fell in the lower half of the data set were considered 'Negative' for code quality concerns. Those in the upper half were considered 'Positive' for code quality concerns. For SQM, there was a natural partition between good and bad reports. As indicated previously, source files with no reports of any kind were excluded from analysis. Classification of source files with mixed reports (see gray column in Figure 3 of section 2.2) are problematic. This concern is directly addressed in the analysis section.

A buffer between the highest and lowest scores was seriously considered. Such a buffer might have been beneficial—even excluding up to one third of the data set—to separate Positive and Negative cases. This middle portion of the data would be considered indeterminate and not scored as a Positive or Negative case. This buffer region was rejected for multiple reasons.

- When the range of the buffer was taken from the aggregate of all data sets, it became clear that some of the software projects had eccentric divisions that skewed the sample heavily toward good or bad quality. For example figure 11 suggests project #4 was almost exclusively bad. There was very little agrement between any of the tools in this respect.

- The exclusion of any portion of the data set affects the breadth of the final error range. Figure 12 shows that the width of that range increases with buffer size. This tends

to distort the confidence interval. The figure also shows that in one case (SonarQube) an excessively large buffer also alters the mean value.

- Figure 12 also reveals that for small buffers (<30%) the error range is relatively stable and comparable to no buffer at all. This stability suggests that no buffer is necessary to obtain a reasonable confidence interval.

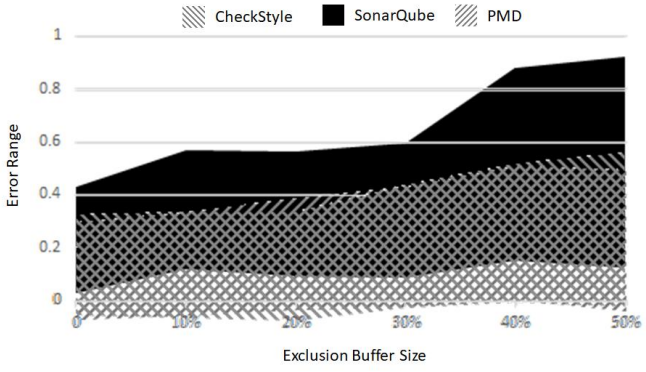


Fig. 12 Variation in error range as a function of buffer size.

A confusion matrix was created using ranked scores from each of the static analysis tools. In the case of SQM, the median rank is mapped to zero. Ranks below the median value are Negative while those above the median are Positive. Metadata (as described in section 2.3) was derived by manual review of the 650 source files. The presence of any such metadata are Positive cases and the absence of metadata is assumed to be a Negative case.

Only 168 instances of metadata were discovered in 109 code files. With just 1.5 instances per file in the True cases, metadata was far from common (<16.8%). The prevalence of metadata, or rather the lack thereof, introduces two concerns for errors. The first is a tendency to assume the False case. That assumption may contribute to a number of False Positives (FPs) and True Negatives (TNs) in the absence of other data. The precision and specificity of the tools will be adversely affected and the likelihood of rejecting the null hypothesis is increased. These conditions are acceptable. If the null hypothesis can be rejected for any tool, a higher burden of proof has been overcome.

The second source of error is the Accuracy Paradox [22] where the TN or True Positive (TP) is overwhelmingly true. Because metadata is so rare, any classifier that always returned a Negative result would still have an accuracy of 58.6%, incorrectly suggesting that the classifier is largely correct. To avoid the paradox, the full confusion matrix was examined with the Diagnostic Odds Ratio (DOR) [8].

The DOR was developed for statistical evaluation of diagnostic tools in cases where a disease is extremely rare and the Accuracy Paradox is a real threat to validity. The DOR is calculated (1) to depend on the whole error matrix, avoiding a bias in favor of False Negatives (FN) or FP. In the DOR range, "higher values indicat[e] better discriminatory test performance" [8]. The DOR has a normal distribution on a logarithmic scale so that standard error (2) and confidence intervals can be calculated (3) for any value of p.

$$DOR = (TP/FP)/(TN/FN) \tag{1}$$

$$Error = \sqrt{1/TP + 1/FP + 1/TN + 1/FN} \tag{2}$$

$$CI = \ln(DOR) \pm Z * Error \tag{3}$$

The DOR has one drawback in that it is undefined (because of a divide by zero error) when TN, FP, or FN is zero. The error may be undefined for similar reasons. To prevent divide-by-zerio errors, it is common practice to seed each value in the confusion matrix. As long as the seed value is much smaller than the other confusion matrix values, the effect of the seed value on DOR is negligible. In this case, a seed value of 1.0 is used—a value far below the sample size N=650 and its distribution in the matrix.

2.5 Null Hypothesis

In this study, there are two areas of interest that have separate null hypotheses:

- H_0 – For each tool in the study, the predictive capability of the tool cannot be statistically distinguished from random chance based on DOR.
- H_1 – For each tool in the study, the predictive capability of the tool cannot be statistically distinguished from the other tools based on DOR.

The first null hypothesis gauges the ability of each tool to predict the presence of metadata that represents True conditions. As noted previously, a lack of exclusionary metadata tends to assume more Negative conditions than may actually exist. Consequently, the number of False Positives and True Negatives may be elevated and there is a higher likelihood of rejecting the null hypothesis. Therefore, it is more likely that H_0 will be rejected for all tools. This threat to the validity is discussed after the results.

The effect on H_1 is less troublesome. While the predictive value of each tool is depressed by the lack of exclusionary metadata, they are all depressed in equal measure. It is possible that one tool may be depressed more than another, but this seems unlikely. None of the tools directly engage in a search for the metadata in question. The results of each tool should be relatively unrelated to the metadata. Evaluation by these criteria should depress each tool to a similar degree. Therefore, comparison of the DOR ranges among the tools should yield more reliable results. This threat to validity is nevertheless discussed after the results.

These hypotheses are considered in light of the research questions posed, rather informally, in section 1 of this paper. The metadata gathered from the sample is representative of results from manual code review. The various static analyzers represent an automated alternative to code reviews. In $\mathbf{H_0}$, we have the means of directly comparing the manual and automated review processes. If we fail to reject $\mathbf{H_0}$ then the tools have not proven more efficient. Rejecting that null hypothesis suggests that the tool is a more efficient mode of reviewing code, at least in some limited area. In $\mathbf{H_1}$, we have the means of comparing the performance of

multiple legacy and novel static code analysis tools. Where $\mathbf{H_1}$ is rejected, the performance of one tool is superior to another.

3 Results

The summary of results are shown in table 1. The full confusion matrix and the two-sided, 95% DOR range are shown for each tool. The table also shows intermediate values for each of the 13 projects, although these values have very wide error ranges. The per-project DOR is simply too inclusive to be meaningful, but that DOR is calculated in each case simply for the sake of completeness.

A close examination of the data may leave the reader with the false impression that the numbers do not tally. Consider that Project #2 only recorded 5 instances of metadata, but the SQM confusion matrix seems to indicate 7 True cases (6 TPs and 1 FN). The apparent discrepancy is caused by the seed values that add 1 to each value (i.e. the actual values are 5 TPs and 0 FNs). In the cumulative data there were no zero values in the confusion matrix, so the seed value was eliminated. Once those seed values are eliminated, the raw values tally.

Relative to Table 1 are figures 13 and 14. These images show the DOR ranges from tabular data in a bar graph. Because the error range on SQM is so large, the critical region is not clear in figure 13. Figure 14 shows that area in greater detail. Note that CheckStyle and PMD barely cross over the line where DOR=1.0, while SonarQube avoids that line with an equally small margin. It is not entirely obvious, even at this scale, but LOC also crosses this critical line. In the case of SQM, however, the confidence interval starts far from this critical value. The significance of this finding will be discussed in the analysis section of this paper.

One final feature of interest in the data set is visible in figure 14. The lower end of the DOR for SQM and upper end of the DOR for SonarQube do in fact touch. It is frustrating that within significant digits the two values are identical. It appears that with additional data sets the relationship would become sufficient to reject H_1.

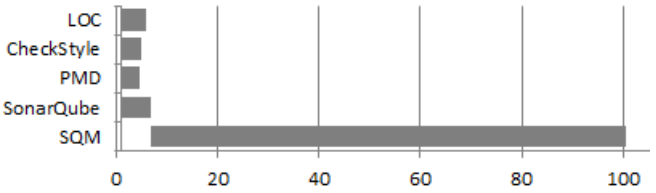


Fig. 13 Full DOR per tool.

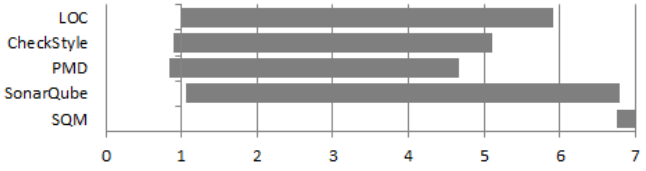


Fig. 14 Critical region of figure 13.

4 Analysis

4.1 Comparison with Random Classification

Recall that H_0 deals with the comparison of each tool with random chance. In the DOR, which exists on a logarithmic scale, values greater than 1.0 show a positive correlation between the classifier and truth data. By the same logic, values less than 1.0 have a negative correlation with truth data. If classification were performed by a random coin flip, DOR would equal 1.0. Therefore, any classifier that crosses the critical line DOR=1.0 cannot be said to be statistically distinct from random chance.

From the available data, it is possible to state with 95% confidence that SonarQube and SQM are statistically distinct from random chance, rejecting H_0 in those cases. LOC, PMD and CheckStyle cannot be be statistically distinguished from random chance and the null hypothesis cannot be rejected. These results are interesting, especially in the context of the correlations charts discussed in section 2.2.

Recall that PMD, CheckStyle, and LOC are highly correlated with each other. Then recall that SonarQube has a

Table 1. Confusion Matrix and DOR by Project and Tool

		SQM						PMD						SonarQube						CheckStyle						LOC					
						DOR Range						DOR Range						DOR Range						DOR Range						DOR Range	
Project	Metadata	TP	FP	TN	FN	min	max	TP	FP	TN	FN	min	max	TP	FP	TN	FN	min	max	TP	FP	TN	FN	min	max	TP	FP	TN	FN	min	max
1	32	11	25	5	2	0.80	794.30	9	18	21	5	0.10	20.00	8	14	25	6	0.10	7.90	9	18	21	4	0.20	25.10	11	15	24	3	0.20	39.80
2	5	6	29	12	1	0.20	1258.90	5	24	28	2	0.10	63.10	5	24	28	2	0.10	63.10	5	25	27	2	0.10	79.40	5	31	21	2	0.10	125.90
3	6	4	21	15	1	0.10	398.10	4	17	29	3	0.00	20.00	4	21	23	3	0.10	31.60	3	18	28	4	0.00	12.60	4	23	21	3	0.10	39.80
4	8	3	39	8	3	0.20	158.50	5	40	7	2	0.50	501.20	5	44	3	2	0.80	1995.30	5	34	13	2	0.20	199.50	5	29	17	2	0.10	125.90
5	10	7	26	5	1	0.50	3162.30	4	15	31	4	0.00	10.00	5	14	32	2	0.00	31.60	6	16	30	2	0.10	39.80	4	12	34	4	0.00	6.30
6	11	6	24	7	2	0.30	316.20	7	15	28	3	0.10	25.10	8	13	30	3	0.10	20.00	5	5	38	6	0.00	2.00	8	11	32	3	0.10	20.00
7	17	6	20	3	4	0.30	316.20	9	10	30	5	0.10	7.90	10	12	28	3	0.10	25.10	9	14	24	4	0.10	15.80	8	13	27	6	0.10	6.30
8	22	8	32	5	1	0.60	3981.10	8	24	18	3	0.30	63.10	9	25	16	2	0.30	158.50	8	26	15	3	0.30	79.40	9	19	23	2	0.20	100.00
9	7	7	19	15	1	0.10	501.20	6	17	25	2	0.10	50.10	6	15	28	2	0.10	39.80	7	22	21	1	0.10	501.20	6	18	26	2	0.10	50.10
10	14	10	24	11	2	0.40	251.20	10	27	13	3	0.40	100.00	11	20	21	2	0.20	125.90	11	25	16	2	0.30	199.50	10	21	20	3	0.20	50.10
11	3	4	26	8	1	0.10	1258.90	4	23	25	1	0.10	316.20	4	21	27	1	0.00	251.20	4	27	22	1	0.10	398.10	4	28	20	1	0.10	398.10
12	2	3	28	8	1	0.10	1000.00	3	28	22	1	0.00	398.10	3	21	29	1	0.00	199.50	3	28	22	1	0.00	398.10	3	30	19	1	0.10	501.20
13	31	14	12	2	3	0.60	1000.00	13	7	22	12	0.00	3.20	16	12	16	9	0.20	10.00	13	7	22	12	0.00	3.20	16	9	20	9	0.10	7.90
Total	168	76	312	91	10	6.76	102.33	74	252	286	33	0.85	4.68	81	243	293	25	1.07	6.76	75	252	286	31	0.89	5.13	80	246	291	28	0.98	5.89

weaker relation with other legacy tools and SQM has no relation at all. The results of this study suggest that those correlations are meaningful. Without hazarding a guess as to the specific aspects of PMD or CheckStyle that cause the correlation with LOC, it seems clear that the correlation indicates poor performance as a classifier. SonarQube and SQM, which correlate poorly with LOC, perform much better as classifiers. At face value, this makes sense as the size of the program should not by itself indicate the quality of the program.

4.2 Comparison among Classifiers

In rejecting H_0, SonarQube and SQM stand alone as classifiers that can predict the software quality to some standard. There is, however, additional evidence that SQM has a further distinction from LOC, PMD, and CheckStyle. With respect to H_1, there can be no statistical distinction between LOC, PMD, CheckStyle, and SonarQube. The DOR for all four are substantially the same. In the comparison between SQM and SonarQube the two ranges just touch, making any distinction statistically insignificant. It can be stated with 95% confidence that SQM is superior to LOC, PMD, and CheckStyle in its predictive capability.

Readers may immediately question the wisdom of publishing the results in this state. Surely, the addition of a 14th or 15th data set would serve to distinguish SQM from SonarQube. Unfortunately, that may not be true. With only three data sets, it was already possible to reject H_0 with references to SQM. Each additional data set, however, brought a variable response in the SQM range—sometimes affecting the mean and sometimes widening the error range. Additional data sets may serve to statistically distinguish the two tools, but the extreme variability of the SQM range is worrisome.

The analysis above is a snapshot of conclusions drawn with a 95% confidence level—conclusions that are invalid at some higher confidence level. Determining the critical Z score that affects H_0 and H_1 reveals how stable these conclusions are. Table 2 shows critical Z scores that allow rejection of H_0 or H_1 with respect to SQM. From this table:

- H_0 is rejected for any of the legacy tools with at least a 90% confidence level.
- H_0 cannot be rejected for SQM below a 99.9% confidence level.
- None of the legacy tools may be statistically distinguished from SQM (i.e. rejecting H_1) at a 99% confidence level. Only SonarQube is statistically indistinguishable from SQM at the 95% confidence level.

4.3 Mismatched Reports Scoring

As described in section 2.2, approximately 25% of all files have mixed reports from SQM. As a result there was some ambiguity as to how to score files in a binary classification. The results reviewed to this point were generated by scoring this group of files as Positive for code quality concerns. This decision was based on an intuitive evaluation that any error reports made the file 'bad code' to some degree.

This section considers alternative scoring for these files.

Table 2. Sensitivity analysis of confidence intervals

	Reject H_0		Reject H_1 w.r.t SQM	
	Z Score	P Value	Z Score	P Value
SQM	3.96	0.001	--	--
SonarQube	1.75	0.041	1.64	0.051
LOC	1.61	0.054	1.75	0.041
CheckStyle	1.42	0.078	1.85	0.033
PMD	1.30	0.097	1.92	0.028

Table 3 reflects the same sensitivity values if the mismatched files were excluded. The option of excluding these files stems from the intuitive understanding that the mixture of information is indeterminant. This option reduces the number of files represented in the SQM analysis and increases the breadth of the DOR. Table 3 shows that H_0 can still be rejected for SQM at a 99% confidence level, but H_1 cannot be rejected for any tool above a confidence level above 85%. These findings suggest that SQM benefits from the error reports in spite of positive quality indicators.

The third scoring option would evaluate mismatched files as good code. Even though this option goes intuitively against the trend observed above, it was considered for completeness. With this alternative scoring, the DOR for SQM crosses that critical DOR=1.0 line. While this scoring does not yield useful results from SQM, it does add further support for believing that any 'bad code' finding from SQM is a Positive case.

Table 3. Repeat of Table 2 excluding mismatched files.

	Reject H_0		Reject H_1 w.r.t SQM	
	Z Score	P Value	Z Score	P Value
SQM	2.37	0.009	--	--
SonarQube	1.75	0.041	0.75	0.227
LOC	1.61	0.054	0.82	0.207
CheckStyle	1.42	0.078	0.93	0.177
PMD	1.30	0.097	0.99	0.162

4.4 Threats to validity

As noted previously, there are two threats to validity. First, a different method of aggregation may (dis)improve the results for any of the tools. Evidence from other papers suggest that with appropriate weights the aggregate values could improve. Recall that Kim [14] used such a set of weights and that these weights effectively nullified the majority of the reports. Even so, Di Nucci [5] replicated a previous study in machine learning and found that with variations in the training data, the results were much poorer. Applying a weighting system to minimize errors from one data set has the potential to discover a local optimum, rather than a generalizable solution.

This may be an area for future work. One may consider individual algorithms, not only SQM but from any tool, as unique detectors. As with Kim [14], this sort of analysis

may lead to a set of credibility ratings or weights to create a metadata tool. If the aggregation method in this study is relatively naïve, a more accurate method must first apply the same analysis at a much finer granularity.

The second threat to validity is the assumption that the absence of metadata (i.e. known 'bad' code) is equivalent to 'good' code. That is not necessarily true, but without metadata that tends to excuse rather than condemn code FPs and TNs are suspect. As discussed previously, this would tend to decrease the mean DOR for all classifiers and increase the chances of rejecting H_0. Note that H_0 could be rejected for PMD, CheckStyle, and LOC at a 90% confidence level. The artificial decrease could account for that much error.

Furthermore, it is assumed that comparing the tools under H_1 is entirely valid as they are equally depressed by the 'good' code assumption. This is not necessarily so. It is possible that one tool is inexplicably depressed more than it should. Refer to figure 14 where the DOR for all legacy classifiers is shown in high detail. If the breadth of the DORs were reduced, even small adjustments to the mean may yield statistical differences between tools.

In both cases, overcoming the assumption of 'good' code may lead to a revision of these results.

5 Conclusions and Future work

In this study, a simple aggregate of all reported findings is used to create binary classifiers from SonarQube, PMD, CheckStyle, and Lines of Code. These classifiers are compared to code quality, as indicated by developer comments and other metadata, using the Diagnostic Odds Ratio. These classifiers are also compared with a novel set of defect detection algorithms knowns as SQM. These tools are compared to random chance in their ability to predict developer satisfaction with code quality. Statistically significant improvements in performance are observed for SQM ($p<.01$) and SonarQube ($p<.05$); PMD, CheckStyle, and Lines of Code cannot be be statistically distinguished from random chance ($p<.05$). The legacy classifiers, (including Lines of Code) are statistically in indistinguishable from each other. SQM performed significantly better ($p<.05$) than PMD, CheckStyle, and Lines of Code as a binary classifier.

Future work in this area falls into three areas. First, increasing the sample of code review files. The larger data set should be published for general review and reuse. Second, apply the same methodology to additional static analysis tools, comparing those results with the present set. Third, apply the same methodology at a finer granularity, determining the degree to which each individual finding is a reliable predictor of code quality. This last investigation may yield insights as the DOR is specifically made for classifiers where the number of positive findings are very low and some of the less verbose algorithms may have a better showing in this paradigm.

The primary weakness of the methodology is a lack of metadata that guarantees 'good' code. The metadata used in the study exclusively identifies 'bad' code and the relative 'goodness' of the remaining code is assumed. Resolving that weakness would affect all future work in this area and yield measurments that are more relaiable.

References

[1] N. Ayewah, W. Pugh, J. Morgenthaler, J. Penix, and Y. Zhou, "Evaluating static analysis defect warnings on production software," in Proceedings of the 7th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering (PASTE), pp. 1–8, ACM, 2007.

[2] S.R. Chidamber and C.F. Kemerer, "A Metrics Suite for Object Oriented Design," IEEE Trans. Software Eng., vol. 20, no. 6, pp. 476-493, 1994.

[3] B.S. Dillon, "Detection of degenerate software forms in object oriented code." U.S. Patent 10,346,287, Jul. 9, 2019.

[4] B.S. Dillon, "A case study in object-oriented patterns identified during manual refactoring." 2016 IEEE 14th International Conference on Software Engineering Research, Management and Applications (SERA). IEEE, 2016.

[5] D. Di Nucci, F. Palomba, D. A. Tamburri, A. Serebrenik, and A. De Lucia, "Detecting code smells using machine learning techniques: are we there yet?" in 2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER). IEEE, 2018, pp. 612–621.

[6] S.G. Eick, T.L. Graves, A.F. Karr, J.S. Marron, and A. Mockus, "Does Code Decay? Assessing the Evidence from Change Management Data," IEEE Trans. Software Eng., vol. 27, no. 1, pp. 1-12, Jan./Feb. 2001.

[7] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1994.

[8] A. Glas, et al. "The diagnostic odds ratio: a single indicator of test performance," Journal of clinical epidemiology, vol. 56, no. 11, pp. 1129-1135, Nov 2003.

[9] R.K. Gupta, et al. "Pragmatic Approach for Managing Technical Debt in Legacy Software Project," in Proceedings of the 9th India Software Engineering Conference. 2016.

[10] T. Holwerda. "WTFs/m" OSnews February 4, 2008 Accessed November 16, 2020. [Online] Available https://mk0osnewswb2dmu4h0a.kinstacdn.com/images/comics/wtfm.jpg

[11] D. Hovemeyer and W. Pugh, "Finding Bugs Is Easy," Proc. Companion to the 19th Ann. ACM SIGPLAN Conf. Object-Oriented Programming Systems, Languages, and Applications, pp. 132-136, 2004.

[12] M. M. Lehman, "Laws of software evolution revisited," in Proceedings of the 5th European Workshop

on Software Process Technology, ser. EWSPT '96. London, UK, UK: Springer-Verlag, 1996, pp. 108–124.

[13] X. Li, "Research on Detection of Abnormal Software Code," in *2nd International Forum on Management, Education and Information Technology Application (IFMEITA 2017)*. Atlantis Press, 2018.

[14] S. Kim and M. D. Ernst. Which warnings should I fix first? In ESEC-FSE '07: Proc. 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering, pages 45–54. ACM, 2007.

[15] D. L. Pamas. Software aging. In Proc. ofthe 16fh Intl. Conf on Software Engineering (ICSE-16), Sorrento, Italy, May 1994.

[16] M. Pizka. Lecture, Topic: "Software Quality – Research Meets Reality" Software Developers' Lecture Series, NWCD Dahlgren, Dahlgren, VA, Nov., 12, 2015.

[17] L.H. Rosenberg, "Applying and Interpreting Object Oriented Metrics," technical report, Software Assurance Technology Center NASA SATC, Apr. 1998.

[18] N. Rutar, C.B. Almazan, and J.S. Foster, "A Comparison of Bug Finding Tools for Java," Proc. IEEE Int'l Symp. Software Reliability Eng. (ISSRE), pp. 245-256, 2004.

[19] J. Spolsky. "Things You should Never do, Part I." Joel on Software. April 6, 2000. Accessed November 16, 2020. [Online] Available www.joelonsoftware.com/2000/04/06/things-you-should-never-do-part-i/.

[20] D. A. Tomassi. Bugs in the Wild: Examining the Effectiveness of Static Analyzers at Finding Real-World Bugs. In Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2018, Lake Buena Vista, FL, USA, November 04-09, 2018, pages 980–982, 2018.

[21] K Trajchevska "Becoming a better developer by using the SOLID design principles" LARACON EU. January 28, 2019. Accessed November 16, 2020. [Online] Available youtube.com/watch?v=rtmFCcjEgEw

[22] F. J. Valverde-Albacete and C. Pelaez-Moreno, "100% classification accuracy considered harmful: The normalized information transfer factor explains the accuracy paradox," PLoS ONE, vol. 9, no. 1, 2014.

[23] J. van Gurp, J. Bosch, and S. Brinkkempter, "Design erosion in evolving software products," Proceedings of the international icsm workshop on the evolution of large-scale industrial software applications (2008), pp. 134-139.

[24] S. Wagner, J. Jürjens, C. Koller, and P. Trischberger. Comparing bug finding tools with reviews and tests. Lecture Notes in Computer Science, 3502:40–55, 2005.

[25] F. Zhang, A. E. Hassan, S. McIntosh, and Y. Zou. The use of summation to aggregate software metrics hinders the performance of defect prediction models. IEEE Transactions on Software Engineering, 43(5):476–491, 2017.