Comparing Static Analysis and Code Smells as Defect Predictors: An Empirical Study

Luigi Lavazza, Sandro Morasca, Davide Tosi

Comparing Static Analysis and Code Smells as Defect Predictors: an Empirical Study

Luigi Lavazza[1][0000−0002−5226−4337], Sandro Morasca[2,3][0000−0003−4598−7024], and Davide Tosi[3][0000−0003−3815−2512]

[1] Università degli Studi dell'Insubria, Varese, Italy luigi.lavazza@uninsubria.it
[2] Università degli Studi dell'Insubria, Como, Italy sandro.morasca@uninsubria.it
[3] Università degli Studi dell'Insubria, Varese, Italy davide.tosi@uninsubria.it

Abstract. *Background.* Industrial software increasingly relies on open source software. Therefore, industrial practitioners need to evaluate the quality of a specific open source product they are considering for adoption. Automated tools greatly help assess open source software quality, by reducing the related costs, but do not provide perfectly reliable indications. Indications from tools can be used to restrict and focus manual code inspections, which are typically expensive and time-consuming, only on the code sections most likely to contain faults. *Aim.* We investigate the extent of the effectiveness of static analysis bug detectors by themselves and in combination with code smell detectors in guiding inspections. *Method.* We performed an empirical study, in which we used a bug detector (SpotBugs) and a code smell detector (JDeodorant). *Results.* Our results show that the selected bug detector is precise enough to justify inspecting the code it flags as possibly buggy. Applying the considered code smell detector makes predictions even more precise, but at the price of a rather low recall. *Conclusions.* Using the considered tools as inspection drivers proved quite useful. The relatively small size of our study does not allow us to draw universally valid conclusions, but our results should be applicable to source code of any kind, although they were obtained from open source code.

Keywords: Defect prediction · code smell · static analysis.

1 Introduction

Software inspections [12, 5, 13, 6] are one of the main techniques that have been proposed for discovering defects in code, to prevent defective software from being released. Software inspections are often performed with the help of checklists, i.e., lists of recurrent issues that usually lead to software failure.

Software bug detectors based on static analysis were developed to automatically recognize code patterns that are generally associated with defects. Bug detectors perform a sort of "automated" inspection, as opposed to the "manual" inspection performed by developers. Unfortunately, static analysis cannot in general provide conclusive evidence of defects. Since many properties related to software defects are undecidable, the indications provided by bug detectors must be verified by developers. In practice, developers have to manually inspect the portions of code that are flagged as possibly defective by tools. Because of their high cost, manual inspections are usually performed only on the sections of code that are considered particularly important or very error-prone. In this sense, bug detectors may be very effective, since they indicate which parts of the code are worth inspecting manually.

The concept of "code smell" was introduced to describe a code structure that is likely to cause problems [36, 8, 16]. The original introduction of the concept of code smell was based on the manual examination of the source code, as witnessed by a few indications. First, Fowler et al. [16] provided only informal descriptions of code smells, since code smells are expected to be easily recognized as inadequate code structures by professional software coders. Second, they did not intend to provide any precise measurement-based definition of code smells (*"In our experience no set of metrics rivals informed human intuition"* [16]). Third, no additional evidence was required that code smells actually have detrimental effects on software. The very same act of code analysis by which a developer recognizes a code smell also lets him/her recognize its harmfulness, hence a situation that is not deemed dangerous is not classified as a code smell, even when the code structurally matches the definition of a code smell.

However, manual code smell detection involves the same type of costs as manual inspections. Therefore, to reduce development costs [23], researchers have developed tools for automatically detecting code smells [28, 9, 43, 29, 50, 44]. Automation was made possible by precise definitions of code smells, generally based on static measures of source code [27].

Even though automatic code smell detectors have been available for a few years, there is little evidence that automatically detected code smells are actually associated with quality issues: there are both reports that support and do not support the association of the presence of code smells with software quality issues. For instance, Olbrich et al. [30] and Palomba et al. [31] reported findings supporting the hypothesis that god class hinders maintainability, while Schumacher et al. [38], Sjøberg et al. [40] and Yamashita et al. [48] reported findings not supporting the hypothesis. Some articles even report cases in which an improvement of software quality in presence of code smells was observed [17].

Other papers studied the correlation between code smells and some structural problems reported by FindBugs, without checking manually whether the detected structural problems correspond to actual defects [11, 51, 41]. In our view, code sections that are classified as smelly by automated detectors should be considered as code sections that need to be manually inspected, to check whether the conditions that could hamper software quality are satisfied.

Given that bug detectors have proved to work reasonably well in detecting real bugs [52, 45, 24], and that manual inspections are expensive, it would be important for software developers to know how well automated bug detectors work by themselves and in combination with code smell detectors. In fact, practitioners who have a given budget for inspections must decide how they should spend it most effectively. Should they favor the indications by bug detectors? Should they inspect the code flagged by automated smell detectors? Or maybe should they proceed to modify the code without inspecting it at all, based exclusively on the indications by the tools? In this paper, we address these questions by investigating the extent of the effectiveness of static analysis as bugs detector by itself and in combination with code smells in guiding inspections. These questions are important especially when developers reuse software written by other developers, as is usually the case with Open Source Software (OSS) [26, 25].

We illustrate an empirical study concerning the OSS products incorporated in two B2C web portals developed by an industrial organization. Our study provides some early findings on the effectiveness of bug detectors aiming at problem detection and automated code smell detection tools. Specifically, we used one tool per category: respectively, SpotBugs [4] and JDeodorant [2]. We applied them to the set of OSS products incorporated in two B2C web portals. We first applied SpotBugs and noted the subset of most important warnings that it issued. We then proceeded to manually check whether those warnings corresponded to actual bugs. Finally, we subjected the code sections related to the warnings to JDeodorant and recorded the smells found.

The main contributions of the paper are the following:

- Given the constant evolution of tools, our study provides some up-to-date evidence about their practical usefulness in software development.
- In our empirical study, we found that the *precision* of the tools in detecting problematic code sections is reasonably high, despite the fact that they do not consider the dynamic behavior of the software under analysis.
- For the first time—to the best of our knowledge—a quantitative evaluation of using bug detectors in combination with code smell detectors is provided.
- We provide some quantitative performance indicators to developers who need to decide how to evaluate the quality of the OSS they are using, or could use, as part of their software.

The results presented in this paper were obtained by applying a process that is close to the one used by practitioners. So, our results are expected to be directly applicable in software development practice.

The remainder of the paper is organized as follows. Section 2 describes the concepts, processes and tools that are the subject of the study. Section 3 describes our empirical study and illustrates the results we obtained. Section 4 discusses the threats to the validity of the study. Section 5 reports about the previous work that aimed at evaluating bug detectors based on static analysis and code smells. Section 6 draws some conclusions.

2 Bug Detectors and Code Smell Detectors

Several tools exist that can be used to reduce inspection costs.

We used two static analysis tools in our empirical study: SpotBugs to automatically detect potential bugs in a software program, and JDeodorant to detect potential code smells and suggest refactoring strategies to improve the source code. These tools were selected because of their characteristics and their diffusion in research and practice. Both tools have very active communities. Both tools are also available as plug-ins, such as for Eclipse or SonarQube platforms.

2.1 A Bug Detector: SpotBugs

SpotBugs [4] is a static analysis tool that looks for bugs in Java source code. The tool is free software, distributed under the GNU Lesser General Public License. SpotBugs inherits all of the features of its predecessor FindBugs [19, 1] and checks more than 400 bug patterns. SpotBugs checks for bug patterns such as—among others—null pointer dereferencing, infinite recursive loops, bad uses of the Java libraries, and deadlocks. SpotBugs is available as an Eclipse plugin at [spotbugs.github.io/eclipse/] or as a standalone program and can be downloaded from [spotbugs.github.io].

In SpotBugs, bug patterns are classified by means of several variables, such as: the type of violation, its category, the rank of the bug, and the confidence of the discovering process.

Ten categories are defined [3], such as "Bad Practice" (i.e., violations of recommended and essential coding practice, like hash code and equals problems, cloneable idiom, dropped exceptions, Serializable problems, and misuse of finalize), "Correctness" (i.e., probable bug - an apparent coding mistake resulting in code that was probably not what the developer intended), or "Multithreaded correctness" (i.e., code flaw issues having to do with threads, locks, and volatiles). The complete list of bug descriptions can be found at [spotbugs.readthedocs.io/en/latest/bugDescriptions.html].

The rank of each warning concerns the severity of the potential bug, and spans from 1 (most severe) to 20 (least severe). Four rank levels are also defined: "scariest" ($1 \leq rank \leq 4$), "scary" ($5 \leq rank \leq 9$), "worrying" ($10 \leq rank \leq 14$), "of concern" ($15 \leq rank \leq 20$).

Moreover, a "confidence" (named "priority" in earlier releases of SpotBugs) is associated to each warning: high confidence (1), normal confidence (2) and low confidence (3), to highlight the quality of the detection process.

2.2 A Code Smells Detector: JDeodorant

JDeodorant [15, 42, 2] is a free tool (available as an Eclipse plug-in) that detects design problems in source code, such as code smells, and suggests how to resolve these smells by applying refactoring procedures. Specifically, JDeodorant is able to detect the following four code smells [16]: God Class (a class that is too long, too complex, and does too much), Long Method (a method, function or procedure that is too large), Type Checking (a class contains "complicated conditional statements that make the code difficult to understand and maintain" [14]), and Feature Envy (a method or an object does not leverage data or methods from its class but asks for external data or methods to perform computation or make a decision).

3 The Empirical Study

3.1 Method

Recently, one of the authors participated in the analysis of the quality of an industrial software product, which used several pieces of OSS [22]. In this study we use those OSS programs, briefly described in Table 1, as a test bed. In practice, the quality of a large fraction of industrial software depends on the quality of OSS. By selecting a set of OSS products that we know are used also in non open-source contexts, we make sure to 1) analyze software that is relevant, and 2) provide results that are of interest also outside the OSS community.

Table 1. The open-source products that were analyzed.

Product name	Version	LOC	Num classes	Num methods
Log4j	1.2.16	16497	217	1910
Jasperreports	6.11.0	278694	2558	23465
Pdfbox	1.8.16	120037	1125	9164
Hibernate-search-elasticsearch	5.11.4	21575	350	3264
Hibernate-search-backend-jgroups	5.11.4	1624	25	1067
Hibernate-search-engine	5.11.4	65239	1020	7967
Hibernate-search-performance-orm	5.11.4	1950	39	159

The study was organized in three phases: data extraction, data analysis, and interpretation of results.

Data extraction was performed as follows:

1. We applied SpotBugs to the set of OSS products. SpotBugs issued several hundred warnings. To limit the effort needed to inspect the code flagged as possibly defective, we considered only the 64 issues having rank not greater than 11. Since SpotBugs ranks warning severity in the range 1–20, we chose 11 as a threshold since it is the upper median of the severity rank range. Considering the issues with the highest ranks is just what developers would do in an industrial setting: having a limited effort to be dedicated to inspections, they deal with the issues classified as most dangerous.
2. The considered issues reported by SpotBugs were inspected manually by the authors. The inspections resulted in classifying every issue as either confirmed (when a real defect was found), rejected (when a false positive was recognized), or "possible" (when we found something wrong, but our knowledge of the code did not allow us to decide whether a failure was actually bound to occur).
3. The code elements (i.e., the classes or methods) involved in the issues reported by SpotBugs were analyzed with JDeodorant, and the detected smells were annotated. We focused on performing smell detection on elements already flagged defective by SpotBugs since, in this paper, we are interested in evaluating the effectiveness of static analysis bug detectors by themselves

and in combination with code smell detectors in guiding inspections, and not *vice versa*.

For instance, in our study, SpotBugs issued a warning of category Correctness and type `rc_ref_comparison` (which is issued when `==` or `!=` operators were used instead of `equals()`) in method `validateEqualWithDefault` of class `Elasticsearch2SchemaValidator`, package `hibernate.search.elasticsearch`. JDeodorant highlighted that class `Elasticsearch2 SchemaValidator` is a smelly class (God Class), and method `validateEqualWithDefault` suffers from Feature Envy. Moreover, manual inspection confirmed that the SpotBugs warning is associated with a real bug.

Data analysis was conducted on the set of code elements flagged as defective by SpotBugs. The analysis was performed twice: once considering the "possible" bugs as false positives, and once considering the "possible" bugs as true positives. In what follows, we label the former scenario as "optimistic" (since the code is less buggy than indicated by SpotBugs), and the latter scenario as "pessimistic" (since the code is considered as buggy as indicated by SpotBugs).

Data analysis was conducted as follows (once for each scenario). First, we computed the number of true positive (TP), false positive (FP), true negative (TN), and false negative (FN) estimates provided by SpotBugs and JDeodorant for both scenarios (see Table 3). In doing this, the existence of a given smell was considered as a fault prediction.

Then, based on TP, FP, TN and FN, we computed a few accuracy indicators, namely *precision*, *recall*, *F-measure* (the harmonic mean of *precision* and *recall*), and ϕ, alias Matthews' Correlation Coefficient (MCC):

$$\begin{aligned}
\textit{precision} &= \tfrac{TP}{EP} = \tfrac{TP}{TP+FP} \\
\textit{recall} &= \tfrac{TP}{AP} = \tfrac{TP}{TP+FN} \\
\textit{F-measure} &= \tfrac{2\ precision\ recall}{precision+recall} \\
\phi &= \tfrac{TP\ TN-FP\ FN}{\sqrt{EP\ EN\ AP\ AN}}
\end{aligned}$$

where EP is the number of estimated positives (EP=TP+FP), EN is the number of estimated negatives (EN= TN+FN), AP is the number of actual positives (AP=TP+FN) and AN is the number of actual negatives (AN=TN+FP).

Finally, we considered two additional smell-based faultiness predictions, which we labeled "any smell" and "all smells." In the former case, a code element is estimated buggy if it has one or more of the smells detected by JDeodorant; in the latter case, a code element is estimated buggy if it has all of the smells detected by JDeodorant. We computed the same accuracy indicators mentioned above for "any smell" and "all smells."

3.2 Results

First of all, let us evaluate the performance of SpotBugs. We considered 64 warnings, hence EP=64. Among these, in the optimistic scenario, TP is the number of confirmed bugs; in the pessimistic scenario, TP is the number of confirmed and possible bugs. Note that we consider only warnings, which—according to SpotBugs—concern potential problems, hence there are estimated positives, but no estimated negatives. Thus, EP=64=n, where n indicates the total number of estimates of the classifier implemented by SpotBugs. As a consequence, we have TN=FN=0, hence TP=AP, and *recall*=1.

The first row of Table 2 summarizes the performance of SpotBugs. Specifically, SpotBugs issued 64 warnings; of these, 37 were recognized via inspections as real bugs, 13 were recognized as false positives, while 14 could not be classified with certainty. Accordingly, in the optimistic case (i.e., when possible bugs are considered as false positives) *precision*=$\frac{37}{64} \simeq 0.54$. In the pessimistic case (i.e., when possible bugs are considered as true positives) *precision*=$\frac{37+14}{64} \simeq 0.8$.

The first row of Table 2 shows that the accuracy of SpotBugs's predictions is good, substantially better than reported in several previous studies (for instance, Shen et al. reported in their study that FindBugs achieved *precision*=40% [39]).

Table 2. SpotBug's issues and *precision.*

Selected		Bugs			*precision*	
issues	n	Confirmed	Possible	Rejected	Optimistic	Pessimistic
all	64	37	14	13	0.58	0.80
high rank	6	6	0	0	1.00	1.00
mid rank	36	13	11	12	0.36	0.67
low rank	22	18	3	1	0.82	0.95
high conf.	22	10	7	5	0.45	0.77
mid conf	42	27	7	8	0.64	0.81

To verify the reliability of the evaluation of the confidence in the warnings, we split SpotBugs issues into high- and mid-confidence ones (there were no low-confidence issues among the ones we considered). We also split SpotBugs issues into high-, mid- and low-rank ones (corresponding to SpotBugs "scariest," "scary," and "worrying" rank levels), to check if the estimation accuracy depends on the rank. The results we obtained are in Table 2. The only noticeable result is that all the 6 high-rank reported issues concern real bugs.

The accuracy indicators for SpotBugs and the considered code smells evaluated by JDeodorant for the optimistic and pessimistic case are given in Table 3. Note that ϕ is undefined for SpotBugs: this is a consequence of EN being zero.

3.3 Interpretation of Results

In interpreting the results, we must take into consideration a few facts:

- For SpotBugs, n=EP, hence FN=TN=0: thus *recall*=$\frac{TP}{AP} = \frac{TP}{TP+FN} = \frac{TP}{TP} = 1$.
- When performing a completely random estimation, you get *precision*=*recall* =*F-measure*=$\frac{AP}{n}$. Therefore, a prediction model having *F-measure* $< \frac{AP}{n}$ should be discarded, since it performs worse than random estimation. In the

Table 3. Accuracy indicators with the optimistic (AP/n=0.58) and pessimistic criterion (AP/n=0.8).

criterion		TP	FP	FN	TN	*recall*	*precision*	FM	ϕ
optimistic	SpotBugs	37	27	0	0	1.00	0.58	**0.73**	—
	GodClass	15	5	22	22	0.41	0.75	0.53	0.23
	LongMethod	22	5	15	22	0.59	0.81	**0.69**	0.41
	FeatureEnvy	5	1	32	26	0.14	0.83	0.23	0.17
	TypeChecking	9	0	28	27	0.24	1.00	0.39	0.35
	AllSmells	2	0	35	27	0.05	1.00	0.10	0.15
	AnySmell	29	9	8	18	0.78	0.76	**0.77**	0.45
pessimistic	SpotBugs	51	13	0	0	1.00	0.80	**0.89**	—
	GodClass	18	2	33	11	0.35	0.90	0.51	0.17
	LongMethod	24	3	27	10	0.47	0.89	0.62	0.20
	FeatureEnvy	6	0	45	13	0.12	1.00	0.21	0.16
	TypeChecking	9	0	42	13	0.18	1.00	0.30	0.20
	AllSmells	2	0	49	13	0.04	1.00	0.08	0.09
	AnySmell	34	4	17	9	0.67	0.89	0.76	0.29

pessimistic case it is $\frac{AP}{n} = 0.8$, while in the optimistic case it is $\frac{AP}{n} = 0.58$. Better than random values of *F-measure* are in bold in Tables 3.

- *The F-measure* has been widely criticized in the literature [18, 34, 49], mainly because it does not account for true negatives. In our case, though, this is not a reason not to use the *F-measure* to evaluate SpotBugs, because TN=0 by construction. As far as code smells are concerned, ϕ complements the *F-measure* in providing a reliable indication of prediction accuracy.

In the pessimistic scenario, no code smell has *F-measure* better than random. The low values of ϕ confirm that in this case code smells are poor defect predictors. However, in the optimistic scenario, Long Method and AnySmell have a *F-measure* better than random, and ϕ confirms that in this case these smells are acceptably good defect predictors. In fact, values of ϕ greater than 0.4 indicate that the association between the defect prediction and model and actual defectiveness is between medium and strong [10].

Nonetheless, in both scenarios, all code smells' *precision* is better than random, and often really good. This is not surprising. Most smells address very specific conditions, which do not occur very frequently. Therefore, they are bound to feature rather low *recall.* On the contrary, when a very specific smell is present, it is expected that there is "something wrong" and a defect is likely present as well. For instance, in the pessimistic scenario, Feature Envy is detected in only 6 cases out of 64, and all 6 code elements were found defective.

3.4 Discussion

SpotBugs appears much more precise than reported by previous—possibly outdated—studies. *Precision* in the [0.58, 0.80] range (depending on "possible" bugs being real bugs or not) suggests that manual inspection of the issues reported by SpotBugs is generally cost-effective. To this end, it is worth noting that SpotBugs describes and localizes possible bugs very precisely. Thus, examining a few lines of code is generally sufficient to recognize the presence of the bug. In many cases, the required correction is also straightforward. So, a first outcome of our analysis is that using SpotBugs to identify the code to be inspected appears cost-effective, even when the evaluated code is OSS, on which industrial developers do not want to invest much effort and time.

However, a practitioner that applied SpotBugs and obtained a set of warnings could still wonder whether SpotBugs warnings are reliable enough to deserve inspections. To clear this doubt, a practitioner could decide to run JDeodorant on the code flagged as possibly defective by SpotBugs, to get further confirmations. Our analyses show (see rows "AnySmell" in Table 3) that this process, which connects static analysis and code smell detection, achieves better *precision* than static analysis by itself, i.e., a greater proportion of inspections find real defects; in other words, inspections are most cost-effective. At the same time, *recall* decreases with respect to inspecting all the warnings issued by SpotBugs; hence, fewer defects are removed.

In conclusion, based on the results of our study, we can suggest that manual code inspection be done following the indications provided by SpotBugs, because its relatively high *precision* level makes it possible to identify (and often correct) several bugs with little effort. However, practitioners may prefer to inspect only code that is flagged as defective by both SpotBugs and JDeodorant; however practitioners are warned that this practice seems to have a slightly increased *precision* and a more substantially decreased *recall.*

Code smells appear characterized by good *precision.* Hence it appears useful to inspect code elements that are classified as smelly. Nonetheless, each inspection could be relatively expensive: for instance, inspecting a God Class involves examining several hundred lines of code. Instead, performing smell detection on elements already flagged defective by bug detectors leads to both increasing the confidence that a smelly piece of code is really defective, and greatly simplifies inspections: in case of a God Class, one does not need to examine the entire class, but only the piece of code flagged defective by the bug detector.

4 Threats to Validity

The external validity of our study may be influenced by the fact that we used only two tools, one for each type of analysis. However, the two tools are among the best-known and most used ones, by both researchers and practitioners. At any rate, we were able to investigate only a few code smells, i.e., all those supported by JDeodorant. So, we may have obtained different results if we had investigated other code smells. Also, we used a limited number of projects and datasets, which may not be representative of a wider section of the software products. In addition, we used OSS projects, which may not be representative of proprietary software products and processes. We limited the number of issues investigated to 64, though we addressed the most critical of a few hundred warnings. As already noted in Section 3.1, we performed smell detection only on elements already flagged defective by SpotBugs, because our goal was not to compare the performance of the two tools in isolation. This is a limitation to the scope of the study, not to its validity; readers are warned not to interpret our results as an evaluation of the performance of code smellers when not used in combination with bug detectors.

Construct validity may be threatened by the performance metrics used. For instance, *FM* has been widely used in the literature, but it also has been largely criticized [49]. We also used *precision*, *recall*, and ϕ, to have a more comprehensive picture about the performance of the tools we used.

5 Related Work

Tools that use static analysis to identify likely defective code have been introduced more than twenty years ago. Several research efforts have been devoted to investigating their real effectiveness.

Rahman et al. [35] compared the defect prediction capabilities of static analysis tools (namely FindBugs, PMD, and Jlint) and statistical defect prediction based on historical data.

Vetrò et al. [47] evaluated the accuracy of FindBugs. The code base used for the evaluation consisted of Java projects developed by students during a programming course. The code is equipped with acceptance tests written by teachers of the course to check all functionalities. To determine true positives, they used temporal and spatial coincidence: an issue was considered related to a bug when it disappeared at the same time as a bug got fixed. Later, Vetrò et al. repeated the analysis, with a larger code set and performing inspections concerning four types of issues found by FindBugs, namely the types of findings considered more reliable [46].

Zazworka et al. studied the relationship between technical debt items reported by developers and the indications provided by FindBugs [52]. They found that FindBugs did well in pointing to source code files with defect debt. However, finer granularity evaluations do not seem to have been addressed.

Danphitsanuphan and Suwantada studied the correlation between code smells and some structural problems reported by FindBugs [11]. However, they did not check whether the structural problems correspond to actual defects.

Zazworka et al. [51] also applied four different technical debt identification techniques (including code smells and automatic static analysis) to 13 versions of an open-source software project. Noticeably, the outputs of the four approaches pointed to different problems in the source code. The research method used by Zazworka et al. [51] is quite different from ours. They looked for correlations between issues reported by tools and actions on code connected with repaying the interests of technical debt. By considering a sufficiently long streak of versions, they obtained a good representation of the underlying relationships between reported issues and the technical debt. Our approach is inherently different. We consider a single version of many software products and manually inspect the code that SpotBugs flags as possibly defective. In this way, we verify whether issues reported by the static analysis tool are actual defects or not.

Thung et al. performed an empirical study to evaluate to what extent field defects could be detected by FindBugs and similar tools [41]. To this end, FindBugs was applied to three open-source programs (Lucene, Rhino and AspectJ). The study by Thung et al. takes into consideration only known bugs. On the contrary, we relied on manual inspection to identify actual bugs.

In 2007, Ayewah et al. evaluated the issues found by FindBugs in production software developed by Sun and Google [7]. They classified the found issues into false positives, trivial bugs, and serious bugs. A substantial fraction of the issues turned out to concern real but trivial problems. Accordingly, they stated that "Trying to devise static analysis techniques that suppress or deprioritize true defects with minimal impact, and high-light defects with significant impact, is an important and interesting research question." 13 years later, we wish to check if SpotBugs (the heir of FindBugs) has improved in detecting "important" issues.

Vestola applied FindBugs to Valuatum's system and found that 18.5% of the issues were real bugs that deserved corrections, 77.6% were mostly harmless bugs, and 3.8% were false positives [45].

Kim and Ernst evaluated the relationship between issues reported by three static analysis tools (including FindBugs) and the history of changes in three open source products [21]. They consider warnings that remain in the programs or are removed during non-fix changes as likely false positive warnings. Although it is probably so, it is hardly so for all such warnings, hence the number of false positives is likely overestimated.

Code smell were defined by Fowler et al. in 1999 [16], based on previous work [36, 8]. A few years later, Marinescu proposed to identify smells on the basis of static code measures [27]: since then, several tools implementing automatic code smell detection—both based on Marinescu's definitions and on other definitions—have been developed, such as Decor, CodeVizard, JDeodorant, etc. [28, 9, 43, 29, 50, 44].

Many researchers addressed the problem of verifying to what extent code smells are associated with code problems that can affect external code qualities (mainly maintainability and correctness). Lately, a few Systematic Literature Reviews (SLR) were published [32, 20, 33, 37], summarizing the evidence collected about code smell harmfulness. The mentioned SLRs depict a situation characterized by several studies, which produced evidence that does not seem conclusive.

6 Conclusions

In this paper, we have described an empirical study that we carried out to assess the usefulness of static analysis and code smell detection in the identification of bugs. Our study uses two popular tools, SpotBugs and JDeodorant, which are applied to a limited set of OSS projects. The study shows that these tools can help software practitioners detect and remove defects in an effective way, to limit the amount of resources that would otherwise be spent in more cost-intensive activities, such as software inspections.

SpotBugs appears to detect defects with good *precision*, hence manual inspection of the code flagged defective by SpotBugs becomes cost-effective. When JDeodorant is used in conjunction with SpotBugs, detection *precision* increases, thus making manual code inspections even more effective. However, *recall* decreases, thus decreasing the number of bugs that are actually identified.

References

1. FindBugs website (2020), http://findbugs.sourceforge.net/
2. JDeodorant website (2020), https://github.com/tsantalis/JDeodorant
3. SpotBugs documentation website (2020), https://spotbugs.readthedocs.io/en/latest/
4. SpotBugs website (2020), https://spotbugs.github.io/
5. Ackerman, A.F., Buchwald, L.S., Lewski, F.H.: Software inspections: an effective verification process. IEEE software **6**(3), 31–36 (1989)
6. Aurum, A., Petersson, H., Wohlin, C.: State-of-the-art: software inspections after 25 years. Software Testing, Verification and Reliability **12**(3), 133–154 (2002)
7. Ayewah, N., Pugh, W., Morgenthaler, J.D., Penix, J., Zhou, Y.: Evaluating static analysis defect warnings on production software (2007)
8. Brown, W.H., Malveau, R.C., McCormick, H.W.S., Mowbray, T.J.: AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis. John Wiley & Sons, Inc., New York, NY, USA, 1st edn. (1998)
9. Codoban, M., Marinescu, C., Marinescu, R.: iproblems-an integrated instrument for reporting design flaws, vulnerabilities and defects. In: 2011 18th Working Conference on Reverse Engineering. pp. 437–438. IEEE (2011)
10. Cohen, J.: Statistical power analysis for the behavioral sciences Lawrence Earlbaum Associates. Routledge, New York, NY, USA (1988)
11. Danphitsanuphan, P., Suwantada, T.: Code smell detecting tool and code smell-structure bug relationship. In: 2012 Spring Congress on Engineering and Technology. pp. 1–5. IEEE (2012)
12. Fagan, M.E.: Design and Code Inspections to Reduce Errors in Program Development. IBM Systems Journal (1976)
13. Fagan, M.E.: Advances in software inspections. In: Pioneers and Their Contributions to Software Engineering, pp. 335–360. Springer (2001)
14. Fokaefs, M., Tsantalis, N., Chatzigeorgiou, A.: Jdeodorant: Identification and removal of feature envy bad smells. In: 2007 IEEE International Conference on Software Maintenance. pp. 519–520 (2007)
15. Fokaefs, M., Tsantalis, N., Chatzigeorgiou, A.: Jdeodorant: Identification and removal of feature envy bad smells. In: 2007 IEEE International Conference on Software Maintenance. pp. 519–520. IEEE (2007)
16. Fowler, M., Beck, K., Brant, J., Opdyke, W., Roberts, D.: Refactoring: improving the design of existing code. Addison-Wesley Professional (1999)
17. Hall, T., Zhang, M., Bowes, D., Sun, Y.: Some code smells have a significant but small effect on faults. ACM Transactions on Software Engineering and Methodology (TOSEM) **23**(4), 1–39 (2014)

18. Hernández-Orallo, J., Flach, P., Ferri, C.: A unified view of performance metrics: translating threshold choice into expected classification loss. Journal of Machine Learning Research **13**(Oct), 2813–2869 (2012)
19. Hovemeyer, D., Pugh, W.: Finding bugs is easy. Acm sigplan notices **39**(12), 92–106 (2004)
20. Kaur, A.: A systematic literature review on empirical analysis of the relationship between code smells and software quality attributes. Archives of Computational Methods in Engineering pp. 1–30 (2019)
21. Kim, S., Ernst, M.D.: Which warnings should I fix first? In: 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering. pp. 45–54 (2007)
22. Lavazza, L.: Software quality evaluation via static analysis and static measurement: an industrial experience. In: The Fifteenth International Conference on Software Engineering Advances – ICSEA 2020. pp. 55–60 (2020)
23. Lavazza, L., Morasca, S., Tosi, D.: An empirical study on the factors affecting software development productivity. e-Informatica Software Engineering Journal **12**(1), 27–49 (2018). https://doi.org/10.5277/e-Inf180102
24. Lavazza, L., Tosi, D., Morasca, S.: An empirical study on the persistence of spotbugs issues in open-source software evolution. In: Int. Conf. on the Quality of Information and Communications Technology. pp. 144–151. Springer (2020)
25. Lenarduzzi, V., Taibi, D., Tosi, D., Lavazza, L., Morasca, S.: Open source software evaluation, selection, and adoption: a systematic literature review. In: 2020 46th Euromicro Conference on Software Engineering and Advanced Applications (SEAA). pp. 437–444 (2020)
26. Lenarduzzi, V., Tosi, D., Lavazza, L., Morasca, S.: Why do developers adopt open source software? past, present and future. In: IFIP International Conference on Open Source Systems. pp. 104–115. Springer (2019)
27. Marinescu, R.: Detection strategies: Metrics-based rules for detecting design flaws. In: 20th IEEE Int. Conf. on Software Maintenance. pp. 350–359. IEEE (2004)
28. Moha, N., Gueheneuc, Y.G., Duchien, L., Le Meur, A.F.: Decor: A method for the specification and detection of code and design smells. IEEE Transactions on Software Engineering **36**(1), 20–36 (2009)
29. Murphy-Hill, E., Black, A.P.: An interactive ambient visualization for code smells. In: 5th international symposium on Software visualization. pp. 5–14 (2010)
30. Olbrich, S.M., Cruzes, D.S., Sjøberg, D.I.: Are all code smells harmful? a study of god classes and brain classes in the evolution of three open source systems. In: 2010 IEEE International Conference on Software Maintenance. pp. 1–10. IEEE (2010)
31. Palomba, F., Bavota, G., Di Penta, M., Fasano, F., Oliveto, R., De Lucia, A.: On the diffuseness and the impact on maintainability of code smells: a large scale empirical investigation. Empirical Software Engineering **23**(3), 1188–1221 (2018)
32. de Paulo Sobrinho, E.V., De Lucia, A., de Almeida Maia, M.: A systematic literature review on bad smells—5 w's: which, when, what, who, where. IEEE Transactions on Software Engineering (2018)
33. Piotrowski, P., Madeyski, L.: Software defect prediction using bad code smells: A systematic literature review. In: Data-Centric Business and Applications, pp. 77–99. Springer (2020)
34. Powers, D.M.: Evaluation: from precision, recall and F-measure to ROC, informedness, markedness and correlation (2011)
35. Rahman, F., Khatri, S., Barr, E.T., Devanbu, P.: Comparing static bug finders and statistical prediction. In: 36th International Conference on Software Engineering. pp. 424–434 (2014)

36. Riel, A.J.: Object-oriented design heuristics, vol. 335. Addison-Wesley Reading (1996)
37. Santos, J.A.M., Rocha-Junior, J.B., Prates, L.C.L., do Nascimento, R.S., Freitas, M.F., de Mendonça, M.G.: A systematic review on the code smell effect. Journal of Systems and Software **144**, 450–477 (2018)
38. Schumacher, J., Zazworka, N., Shull, F., Seaman, C., Shaw, M.: Building empirical support for automated code smell detection. In: ACM-IEEE international symposium on empirical software engineering and measurement. pp. 1–10 (2010)
39. Shen, H., Fang, J., Zhao, J.: Efindbugs: Effective error ranking for findbugs. In: 2011 Fourth IEEE International Conference on Software Testing, Verification and Validation. pp. 299–308. IEEE (2011)
40. Sjøberg, D.I., Yamashita, A., Anda, B.C., Mockus, A., Dybå, T.: Quantifying the effect of code smells on maintenance effort. IEEE Transactions on Software Engineering **39**(8), 1144–1156 (2012)
41. Thung, F., Lo, D., Jiang, L., Rahman, F., Devanbu, P.T., et al.: To what extent could we detect field defects? an extended empirical study of false negatives in static bug-finding tools. Automated Software Engineering **22**(4), 561–602 (2015)
42. Tsantalis, N., Chaikalis, T., Chatzigeorgiou, A.: JDeodorant: Identification and removal of type-checking bad smells. In: 2008 12th European Conference on Software Maintenance and Reengineering. pp. 329–331. IEEE (2008)
43. Tsantalis, N., Chatzigeorgiou, A.: Identification of extract method refactoring opportunities for the decomposition of methods. Journal of Systems and Software **84**(10), 1757–1782 (2011)
44. Van Emden, E., Moonen, L.: Java quality assurance by detecting code smells. In: Ninth Working Conference on Reverse Engineering. pp. 97–106. IEEE (2002)
45. Vestola, M., et al.: Evaluating and enhancing findbugs to detect bugs from mature software; case study in valuatum (2012)
46. Vetrò, A., Morisio, M., Torchiano, M.: An empirical validation of findbugs issues related to defects. In: 15th Annual Conference on Evaluation and Assessment in Software Engineering (EASE 2011). pp. 144–153. IET (2011)
47. Vetrò, A., Torchiano, M., Morisio, M.: Assessing the precision of FindBugs by mining java projects developed at a university. In: 2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010). pp. 110–113. IEEE (2010)
48. Yamashita, A.: Assessing the capability of code smells to explain maintenance problems: an empirical study combining quantitative and qualitative data. Empirical Software Engineering **19**(4), 1111–1143 (2014)
49. Yao, J., Shepperd, M.J.: Assessing software defection prediction performance: why using the matthews correlation coefficient matters. In: EASE '20: Evaluation and Assessment in Software Engineering, Trondheim, Norway, April 15-17, 2020. pp. 120–129. ACM (2020)
50. Zazworka, N., Ackermann, C.: Codevizard: a tool to aid the analysis of software evolution. In: Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement. pp. 1–1 (2010)
51. Zazworka, N., Izurieta, C., Wong, S., Cai, Y., Seaman, C., Shull, F., et al.: Comparing four approaches for technical debt identification. Software Quality Journal **22**(3), 403–426 (2014)
52. Zazworka, N., Spínola, R.O., Vetrò, A., Shull, F., Seaman, C.: A case study on effectively identifying technical debt. In: Proceedings of the 17th International Conference on Evaluation and Assessment in Software Engineering. pp. 42–47 (2013)