Assessing the Precision of FindBugs bymining Java Projects developed at a University

Assessing the Precision of FindBugs by mining Java Projects developed at a University

Antonio Vetro', Marco Torchiano, Maurizio Morisio
Politecnico di Torino
Torino, Italy
name.surname@polito.it

***Abstract*—Software repositories are analyzed to extract useful information on software characteristics. One of them is external quality. A technique used to increase software quality is automatic static analysis, by means of bug finding tools. These tools promise to speed up the verification of source code; anyway, there are still many problems, especially the high number of false positives, that hinder their large adoption in software development industry. We studied the capability of a popular bug-finding tool, FindBugs, for defect prediction purposes, analyzing the issues revealed on a repository of university Java projects. Particularly, we focused on the percentage of them that indicates actual defects with respect to their category and priority, and we ranked them. We found that a very limited set of issues have high precision and therefore have a positive impact on code external quality.**

Keywords: Software Quality, Automatic Static Code Analysis, Defect prediction, Bug Finding Tools

I. INTRODUCTION

Software quality assurance is a very critical activity: it is historically estimated that rework effort is about 40-50% of the whole software production effort [4] [5]. Several techniques can be used to improve quality, we focus on automatic static analysis and particularly on bug finding tools. Bug finding tools analyze the source code by applying a set of rules and produce a list of issues corresponding to violations of the rules. The issues are supposedly defects of the program that ought to be removed or fixed.

The software engineering literature still lacks a thorough assessment of bug finding tools, and many problems have been identified in literature:

- high number of false positives [15][11]
- detection of only a reduced subset of possible bugs [15][16]
- the efficiency of the default issues prioritization decided by tool's author [10][2]
- the dubious economical benefits brought by their usage [14][16] .

We studied one of these problems: precision of issues revealed by bug findings tools. Our goal is to answer the following research question: *which issues are actual predictors of bugs, and which are not?* This knowledge is very important to provide the developers with accurate information that can be used effectively in developing and maintaining the software.

We conducted an empirical validation of the issues of a widely used tool: FindBugs v1.3.8. In particular we analyzed the issues produced by FindBugs on a large pool of similar programs. The main contributions of the paper are:

- It provides empirical evidence about the validity of issues categories as bug predictors;
- As a consequence identifies a first step to make bug-finding tool usage more effective;
- Using a large pool of developers, it eliminates the effect of developer style on the results.

II. CONTEXT AND DEFINITIONS

The program pool was developed in the context of the Object Oriented Programming (OOP) course at the authors' university, where students develop Java programs for the exam. Students develop a first version of the program in laboratory (the "lab" version), then a tool, PoliGrader[13], manages the delivery process and runs a suite of black box acceptance tests (JUnit classes): results of tests and their source code are sent back to the students, that go home and improve the lab version, creating a version of the program, called "home" version, that must pass all acceptance tests.

The code base used in the experiment consists of 85 Java assignments from the 2009 OOP course: requirements are the same for all the assignments; and they are publicy available at the following URL: http://softeng.polito.it/vetro/confs/msr2010/Requirements.htm. Each assignment contains both lab and home versions syntactically correct, and home version passes 100% of the acceptance tests. Acceptance tests are written by teachers of the course in such a way all functionalities are checked. Teachers develop also a correct "solution program", and they check test coverage on it. The average size of projects is 166.4 NCSS (Non Commenting Source Statements) for lab versions and 183.81 NCSS for home versions. The estimated number of function points for the project is 66.30.

An issue produced by FindBugs is characterized by an ID, a textual explanation, and a location in the source code. The issues are categorized by FindBugs according to two

dimensions: category (Bad Practice, Correctness, Style, Performance, and Malicious Code are the categories with at least one issue signaled in our code base) and priority (Low, Medium, High). Both classifications have been decided by the tool's authors and are based on their personal experience.

III. EXPERIMENT DEFINITION

To address the research question we consider a main *dependent measure*: precision of the issues that can be defined as the proportion of the signaled issues that correspond to actual defects.

Precision is a derived measure that can be computed on the basis of the following primitive measures: *NI*, the number of issues signaled by FindBugs and *NA*, the number of issues corresponding to actual defects. We do not compute recall (commonly coupled with precision), because it would require the knowledge of the complete set of defects. This can be computed only by hand: given the large number of projects to be checked this is a long and error prone process.

To determine *NA* we adopted the concepts of temporal and spatial coincidence, previously presented in literature in [6] [10] [7]. We have temporal coincidence when one or more issues disappear in the evolution from the lab to the home version, and in the same time one or more defects are fixed: probably those issues were related to the fixed defects. In this context defects fixed are revealed when a test that in lab version fails instead in home version succeeds.

The possibility that a disappearing issue was not related to the disappearing defect is the noise of this metric, that is filtered out by adding spatial coincidence: we observe spatial coincidence when an issue's location corresponds to lines in the source code that have been modified in the evolution from the lab to the home versions.

In practice, the combination of temporal and spatial coincidence is interpreted as a change intended to remove the issue, that is linked to the defect.

The procedure followed to conduct the study is very simple: we ran the FindBugs tool on both versions of each assignment in the repository, then we collected the information about the change performed to evolve the lab version into the home version. The changes were identified using the DiffJ tool, which operates on two versions of a Java program and is able to compute for each pair of corresponding Java classes which lines changed.

Afterwards, we computed precision of issues, first without considering categories and priorities, then analyzing results observing each issue group (combination of category and priority) separately.

To determine whether an issue group is a good or bad defect predictor, we established 2 precision thresholds and we performed statistical test against null hypotheses. Thresholds were established after observing the distribution of issues precision for each assignment (Table I and Figure 1), without distinction of categories and priorities.

TABLE I. PRECISION OF THE WHOLE SET OF ISSUES

Min	1st Q	Median	Mean	3rd Q	Max	St dev
0	0	0	0.149	0.25	0.8	0.226

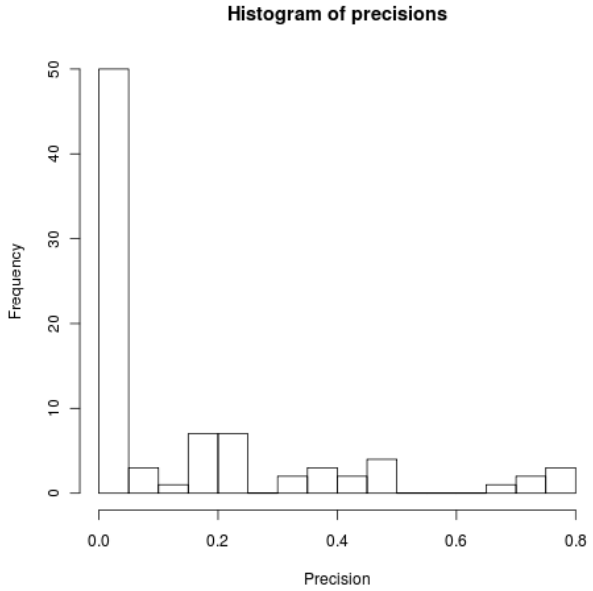

Figure 1. Hystogram of precisions

The mean of precisions is quite low (0.15) and the variability is high. We decided to consider the issue group (group G in the following) as a defect predictor if it has a precision greater than 30%. Such a low value is justified by the exploratory nature of this work and it compensates for the large variability we expect to find in each group. Furthermore this value is far enough from the average precisions of the issues: in 50% of assignments precision is 0; in 75% (3rd quartile) of the assignments precision is at most 0.25, less than the threshold; finally, the 30% precision threshold is the double of the mean of precisions, that is a quite wide ratio.

To identify the issue groups that can be considered as defect predictors, we define the first null hypothesis:

HA_0: *precision of the issues belonging to group G is less than 30%.*

The next step is to find false positives, the bad defects predictors. We consider as false positives the ones with precision <5%, a very low threshold. So we formulate the following parametric null hypothesis:

HB_0: *precision of the issues belonging to group G is grater than 5%.*

Read together, the two hypotheses mean that a group of issues G is a good predictor (GP) if precision of the issues

that it contains is >30% and is a bad predictor (BP) (i.e. a generator of false positives) if precision of the issues that it contains is <5%. The goal of the data analysis is to reject the above null hypothesis by means of statistical tests. For this purpose we selected the single-tailed proportion test with binomial distribution [12]. Given a sample proportion and sample size, such a test computes the probability that the general population (from which the sample is extracted) has a proportion greater (or lower) than a reference proportion. To reject the null hypothesis we adopt the standard significance level at 5%, that is the probability of rejecting a null hypothesis when it is true (type I error) we consider acceptable.

IV. RESULTS

Overall FindBugs revealed a total of 508 issues (NI) in the 85 lab versions of the assignments, among them 94 (NA) were removed in changed lines (temporal and spatial coincidence). Table II shows NA / NI at issue group level. Table III contains precisions and hypothesis tests computed for each different issue group (p-values are shown below precision). Columns of Table II and Table III contain abbreviations of the full names of categories, that are: Bad Practice, Correctness, Malicious Code, Performance, Style.

The full tables with number of detections (NI) and number of issues removed in changed lines (NA) for each project and each issue group are available at the following URL: http://softeng.polito.it/vetro/confs/msr2010/ .

TABLE II. DETECTIONS

	Bad Pr.	**Corr.**	**Mal.C.**	**Perf.**	**Style**
Low	5 / 70	1 / 3	0 / 0	0 / 7	5 / 11
Medium	2 / 145	12 / 45	4 / 15	31 / 144	6 / 16
High	13 / 28	12 / 19	0 / 0	0 / 0	3 / 5

TABLE III. PRECISION:TEMPORAL + SPATIAL COINCIDENCE

	Bad Pr.	**Corr.**	**Mal.C.**	**Perf.**	**Style**
Low	7%	33%	NA	0%	45%
HA	1	0.50	NA	0.91	0.21
HB	0.71	0.82	NA	0.50	1
Medium	**1%**	27%	27%	22%	38%
HA	1.00	0.63	0.50	0.98	0.35
HB	**0.04**	1	1	1	1
High	46%	63%	NA	NA	60%
HA	**0.05**	**<0.01**	NA	NA	0.16
HB	1	1	NA	NA	1

HA: The null hypothesis is rejected only for categories Bad Practice and Correctness both at High priority: this is the set of true positives for spatial + temporal coincidence. All the other groups have non significant p-values and exhibit low estimate precisions except for Style at High priority which has a relatively high precision, though not significant.

HB: Bad Practice and Performance at Low priority, and Bad Practice Medium priority, are the groups whose precision is lower then 5%: however, only Bad Practice at Medium priority has a significant p-value, and we can reject HB_0 for this group.

V. DISCUSSION

The results from the hypothesis testing presented above let us identify the sets of good and bad defect predictor issue groups.

On the basis of these results, we built a partial ordering of the issue groups dividing them into three sets: *good*, *bad* and *ambiguous*. We devised the ordering by putting in the set of good issues the issues marked as defect predictors, in the set of bad issues those issues marked as false positives, and in the set of ambiguous issues all the others that haven't been classified . The set of good predictor issues is GP={*Bad Practice High*, *Correctness High*}, the set of bad predictors is BP={*Bad Practice Medium*}, and the remaining issue groups are ambiguous. Counting the single issues belonging to those groups, they are just 8 out of 359 (2.23 %).

The rationale of this ranking is a new prioritization of warnings based on groups, that takes into account the probability of signaling a defect. An important practical application of this finding is a filtering strategy that can avoid to developers the information overload constituted by a very large number of issues: in our datasets bad predictor issues are the 28.5 % of the total detections in lab versions. Fixing issues with a low probability of being related to a defect is dangerous since we know from Adam's law [1] that the probability of introducing a new error during a fault correction is always different from zero.

VI. THREATS TO VALIDITY

We can identify 2 threats: an external and a construct threat.

The external threat is: we have studied small student projects, hence the application of findings in industrial context is debatable.

Construct threats is concerning the identification of defects. In this study, no bug database was available: we made the assumption that all changes were done to fix a defect: actually, it is possible that some changes were not related to real defects, but to other motivations (cleaner code, more readable code, and so on). Nevertheless, we don't expect that this kind of noise could change results and ranking, because usually students correct the lab versions in a *quick and dirty* way, doing as few changes as possible, for two reasons: 1) the home version is the last version of the

project, actually no maintenance has to be done subsequently; 2) students are discouraged in doing many changes, because the mark suggested by PoliGrader decreases with the quantity of changes made (see details in [13]).

VII. Related Work

As already mentioned in section 3, temporal and spatial coincidence have been used by Boogerd and Moonen [6] and by Kim and Ernst [10]. Our research confirmed the findings of [6]: a reduced set of rule violations (even smaller in percentage, almost identical in absolute value) has impact on code quality. Difference with their findings is that our "bad issues" are less then their "bad violations". Further, our good issues set is composed exclusively by high priorities issues, and our bad issues set exclusively by medium priority: default prioritization of issues seems to be effective, in contrast with what is found in [10] (but not in University context).

Looking at other studies specifically related to FindBugs ([3], [9] and [8]), manual checks of issues brought to high percentages of true positives: overall percentages declared are always higher than 50 %.

On the other side, a study by Wagner et al. [14] demonstrated that FindBugs and PMD (another bug finding tool) were able to find only the 16% of defects in one project, and none in another one. Our study is the first differentiating assessment of issues precision by category and priority, and the first that eliminates the effect of the developer style since a large pool of developers developed the same software.

VIII. Conclusions and Further Work

The analysis of precisions demonstrated that only 2 out of 15 groups of issues can be considered as reliable predictors of actual defects, and one group of issues has a precision that is practically negligible. These findings and the adoption of the technique used may have a practical impact in filtering issue notifications for developers to reduce information overload. Future work will be devoted to: repeat temporal and spatial analysis with higher level of detail, specifying the single issues, besides categories and priorities, and study the possible correlation between groups of issues.

References

[1] Edward N. Adams. Optimizing preventive service of software products. *IBM Journal of Research and Development*, 28(1):2–14, 1984.

[2] Nathaniel Ayewah, David Hovemeyer, J. David Morgenthaler, John Penix, and William Pugh. Using static analysis to find bugs. *IEEE Software*, 25(5):22–29, 2008.

[3] Nathaniel Ayewah, William Pugh, J. David Morgenthaler, John Penix, and YuQian Zhou. Evaluating static analysis defect warnings on production software. In *PASTE '07: Proceedings of the 7th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 1–8, New York, NY, USA, 2007. ACM.

[4] B. W. Boehm. Software process management: lessons learned from history. In *ICSE '87: Proceedings of the 9th international conference on Software Engineering*, pages 296–298, Los Alamitos, CA, USA, 1987. IEEE Computer Society Press.

[5] Barry Boehm and Victor R. Basili. Software defect reduction top 10 list. *Computer*, 34(1):135–137, 2001.

[6] C. Boogerd and L. Moonen. Assessing the value of coding standards: An empirical study. *Software Maintenance, 2008. ICSM 2008. IEEE International Conference on*, pages 277–286, 28 2008-Oct. 4 2008.

[7] Cathal Boogerd and Leon Moonen. Evaluating the relation between coding standard violations and faultswithin and across software versions. *Mining Software Repositories, International Workshop on*, 0:41–50, 2009.

[8] Brian Cole, Daniel Hakim, David Hovemeyer, Reuven Lazarus, William Pugh, and Kristin Stephens. Improving your software using static analysis to find bugs. In *OOPSLA '06: Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*, pages 673–674, New York, NY, USA, 2006. ACM.

[9] David Hovemeyer, Jaime Spacco, and Bill Pugh. Evaluating and tuning a static analysis to find null pointer bugs. Lisbon, Portugal, September 5–6, 2005. ACM.

[10] Sunghun Kim and Michael D. Ernst. Which warnings should i fix first? In *ESEC-FSE '07: Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 45–54, New York, NY, USA, 2007. ACM.

[11] Zhenmin Li, Lin Tan, Xuanhui Wang, Shan Lu, Yuanyuan Zhou, and Chengxiang Zhai. Have things changed now? An empirical study of bug characteristics in modern open source software. In *ASID '06: Proceedings of the 1st workshop on Architectural and system support for improving software dependability*, October 2006.

[12] Erkki P. Liski. An introduction to categorical data analysis, 2nd edition by alan agresti. *International Statistical Review*, 75(3):414–414, December 2007.

[13] Maurizio Morisio and Marco Torchiano. A fully automatic approach to the assessment of programming assignments. *International Journal of Engineering Education*, 0(0):1–16, 2009.

[14] S. Wagner, F. Deissenboeck, M. Aichner, J. Wimmer, and M. Schwalb. An evaluation of two bug pattern tools for java. In *Software Testing, Verification, and Validation, 2008 1st International Conference on*, pages 248–257, 2008.

[15] Stefan Wagner, Jan Jurjens, Claudia Koller, Peter Trischberger, and Technische Universitat Munchen. Comparing bug finding tools with reviews and tests. 2008.

[16] J. Zheng, L. Williams, N. Nagappan, W. Snipes, J. P. Hudepohl, and M. A. Vouk. On the value of static analysis for fault detection in software. *Software Engineering, IEEE Transactions on*, 32(4):240–253, 2006.