An Empirical Validation of FindBugs Issues Related to Defects

An Empirical Validation of FindBugs Issues Related to Defects

Antonio Vetro', Maurizio Morisio, Marco Torchiano
Politecnico di Torino
first.last@polito.it

Abstract

***Background**: Effective use of bug finding tools promise to speed up the process of source code verification and to move a portion of discovered defects from testing to coding phase. However, many problems related to their usage, especially the large number of false positives, could easily hinder the potential benefits of such tools.*

***Aims**: Assess the percentage and type of issues of a popular bug- finding tool (FindBugs) that are actual defects.*

***Method**: We analyzed 301 Java Projects developed at a university with FindBugs, collecting the issues signalled on the source code. Afterwards, we checked the precision of issues with information on changes, we ranked and validated them using both manual inspection and validation with tests failures.*

***Results**: We observed that a limited set of issues have high precision and conversely we identified those issues characterized by low precision. We compared findings first with our previous experiment and then to related work: results are consistent with both of them.*

***Conclusions**: Since our and other empirical studies demonstrated that few issues are related to real defects with high precision, developers could enable only them (or prioritize), reducing the information overload of FindBugs and having the possibility to discover defects earlier. Furthermore, the technique presented in the paper can be adopted to other tools on a code base with tests to find issues with high precision that can be checked on code in production to find defects earlier.*

1. Introduction

Automatic static analysis (ASA) is performed on source code with different goals: improve important characteristics of code (such as maintainability), check a standard compliance or detect possible defects. We call Bug Finding tools those ASA tools whose principal goal is to identify defects on source code. Bug Finding tools look for violations of reasonable and recommended programming practices, bug and design patterns, and they are able to automatically list all violations (we call them *issues*, that are supposedly defects of the program that ought to be removed), statically analyzing source code or intermediate code (at compile time). They are easy to use (it is just a matter of running the main and check the output), they are scalable (they can analyze thousands of lines of code in few minutes). Furthermore, Bug Finding tools promise to speed up the verification process because they evaluate software in the abstract, without running it or considering a specific input. Another possible advantage in terms of time is that these tools do not need a working code base, contrary to the other usual VV activities like testing and code inspections that have hence a consistent delay injection. Given that the longer the delay of a fault insert-remove is, the higher the cost of removing that defect is (Boehm and Basili(2001)), the introduction of Bug Finding tools in VV process, and especially in production code, could lead to important economical benefits.

However, despite all the advantages we have listed, several problems and obstacles to the adoption of these tools were identified in the literature in the last years:

- high number of false positives (Wagner et al.(2005)Wagner, Jürjens, Koller, Trischberger, and München) (Li et al.(2006)Li, Tan, Wang, Lu, Zhou, and Zhai)
- detection of a reduced subset of possible bugs only (Wagner et al.(2005)Wagner, Jürjens, Koller, Trischberger, and München) (Zheng et al.(2006)Zheng, Williams, Nagappan, Snipes, Hudepohl, and Vouk)
- dubious efficiency of the default issues prioritization decided by tool's author (Kim and Ernst(2007))
- questionable economical benefits brought by their usage (Wagner et al.(2008)Wagner, Deissenboeck, Aichner, Wimmer, and Schwalb)

(Zheng et al.(2006)Zheng, Williams, Nagappan, Snipes, Hudepohl, and Vouk)

We focus our research on the first problem. In our previous work (Vetro' et al.(2010)Vetro', Torchiano, and Morisio), we analyzed the issues produced by FindBugs v1.3.8 (Hovemeyer and Pugh(2004)) on a pool of 85 similar small programs, each of them developed by a different student in our university. The goal of our experiment was to verify which FindBugs issues were related to real defects on source code and which not. In the work we present in this paper we reproduce the same experiment, with the following improvements: we enlarge the code base (301 projects), we consider the single FindBugs issues instead of considering only the categories and we use functional tests failures to validate the relationship FindBugs between issues and defects in the code. The knowledge of the issues related to defects is very important to provide the developers with accurate information that can be used effectively in developing and maintaining the software.

We describe the context in which the experiment is conducted in Section 2. Then we define the experiment design and discuss threats to validity in Section 3. In Section 4 we show results and their validation, then in Section 5 we discuss the results, comparing them with previous and related work. We conclude in Section 6 summarizing our findings and contributions to the state of the art and we introduce the future work.

2. Experiment Context

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. The exam procedure is carried out on six steps.

1) Teachers define the project and provide the students with a textual description and a set of wrapper classes. The students develop a first version of the program in the laboratory (the "lab" version) and submit it to a central server by means of an Eclipse Plugin.

2) A tool on the server, PoliGrader (Torchiano and Morisio(2009)), manages the delivery process and runs a suite of black box acceptance tests (JUnit classes). Acceptance tests are written by teachers of the course to check all functionalities required and the highest possible code coverage is obtained running tests on a correct solution program.

3) Results of test execution and test source code are sent back to the students.

4) Students improve the lab version at home, creating a new version of the program (the "home" version), that must pass all acceptance tests. This new version is submitted back to the server.

5) The PoliGrader tool checks that home versions pass all tests and compute marks taking in considerations the numbers of tests passed in the lab version and the diff between lab and home version.

6) All information (marks, source code, tests, and changes) is available to teachers in order to finally evaluate the students.

The code base used in the experiment consists of 301 Java projects from 7 different exam sessions: requirements are the same for all projects belonging to the same session. Each project contains both lab and home versions syntactically correct, the mean size of projects is about 200 non commented source statements (NCSS), each project containing from 4 up to 9 Java classes. As anticipated above, the issues reported by FindBugs are violations of rules of correct programming or bug patterns in the source code that could be related to real defects: if so, we call them "good defect predictors", otherwise "bad defect predictors". Moreover, the same issue can be detected in different places of the code: we call them occurrences or detections.

3. Experiment Design

Adhering to the Goal-Question-Metric approach (Basili et al.(1994)Basili, Caldiera, and Rombach) we first define the goal of the research at conceptual level, which is formally presented in Table1. The goal aims at identifying the issues revealed by FindBugs and their relationship with the defects. Corresponding to the goal we formulate the research question (RQ1) and identified the relative metric (M1).

Table 1. Goal of the study

	Goal
Purpose	Identify and characterize
Issue	issues linked to real defects and generated
Object (Process)	by FindBugs 1.3.8 analysis on 301 University Java Projects
Viewpoint	from the view point of a student Java programmer

- RQ1: Which FindBugs issues are related to defects (good defect predictors) and which not (bad defect predictors)?
- M1: Issue precision (spatial + temporal coincidence)

To address research question RQ1 we consider a main dependent measure: the precision of the issues (M1) that can be defined as the proportion of the signaled issues that correspond to actual defects. The 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. To determine NA we adopted the concepts of temporal and spatial coincidence, previously presented in literature in (Boogerd and Moonen(2008)) and (Kim and Ernst(2007)), and used also in our previous work (Vetro' et al.(2010)Vetro', Torchiano, and Morisio). 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. Figure 1 and Figure 2 show a real example of temporal coincidence, extracted from the programs examined with FindBugs in the experiment. We observe in Figure 1 that an issue (self assignment of a field) is signaled on line 9: the field `forum` is assigned to it self. In the evolution from lab to home version (Figure 2) the student discovers the error and adds a parameter to the constructor's method, in such a way it is assigned to the field `forum`. The issue effectively disappears in the home version. However, the real cause of the fault isn't on line 9, but on the list of parameters on lines 1-2-3: in fact the student modified only line 3 (underlined in Figure 2).

Hence, there is a possibility that a disappearing issue is not related to the disappearing defect: this is the noise of temporal coincidence metric that can be filtered out by adding the 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. Figure 3 and Figure 4 show an example of temporal + spatial coincidence. In the lab version (Figure 3), an issue is signaled on line 6: it is an infinite recourse loop, because the function calls itself without any stopping criterion. In the new version (Figure 4), the student detects the error and fixes it changing line 6 (underlined): in the home version the issue is no longer signaled and it was located in the same line changed during the fix, therefore we observe temporal + spatial coincidence. In practice the combination of temporal and spatial coincidence is interpreted as a change intended to remove the issue, which is linked to a defect. After the computation of precision with temporal + spatial coincidence method, we establish 2 precision thresholds and we perform a statistical test against null hypotheses to determine whether an issue is a good or bad defect predictor.

```
1. public User (String nick, String
2. first, String last, String email, String
3. password) {
4. this.nick = nick ;
5. this.first = first ;
6. this.last = last ;
7. this.email = email ;
8. this.password = password;
9. this.forum = forum;
10.     }
```

Figure 1. Temporal coincidence. Lab Version

```
1. public User (String nick, String
2. first, String last, String email, String
3. password, Forum forum) {
4. this.nick = nick ;
5. this.first = first ;
6. this.last = last ;
7. this.email = email ;
8. this.password = password;
9. this.forum = forum;
10.     }
```

Figure 2. Temporal coincidence. Home Version

```
1. public Researcher getResearcher
2. (String id) throws NoResearcherException
3. {
4. if (! this.resarchermap.containsKey(id))
5. throw new NoResearcherException();
6. return this.getResearcher(id); ! infinite recurse loop !
7. }
```

Figure 3. Spatial + temporal coincidence. Lab Version

```
1. public Researcher getResearcher
2. (String id) throws NoResearcherException
3. {
4. if (! this.resarchermap.containsKey(id))
5. throw new NoResearcherException();
6. return this.researchmap.get(id);
7. }
```

Figure 4. Spatial + temporal coincidence. Home Version

The 2 thresholds are:

- a minimum precision threshold that issue must exceed to be considered as good defect predictor,
- a maximum precision threshold that issues must not exceed to be eligible to the role of bad defects predictors.

Given the exploratory nature of this work, we decide to consider an issue as a good defect predictor if it has a precision greater or equal to 50%. Such threshold is also a compromise between the different true positive ratios of FindBugs issues found in literature, and it is higher than the threshold used in (Vetro' et al.(2010)Vetro', Torchiano, and Morisio) because we want to achieve stronger results. Therefore, we can formulate the first null hypothesis as follows:

HA_0: *the precision of issue I is not greater than 50%.*

The next step is to find false positives, i.e. bad defects predictors. We consider as bad defects predictors those issues with precision <=5%, a very low threshold, that we consider a strict inclusion criterion. So we formulate the following null hypothesis:

HB_0: *the precision of issues I is not lower than 5%.*

Read together, the two hypotheses mean that an issue I is a good predictor (GP) if hypothesis HA_0 can be rejected, i.e. its precision is >=50%, conversely it is a bad predictor (BP) (or source of false positives) if hypothesis HB_0 can be rejected, i.e. its precision is <=5%. The goal of the data analysis is to reject the above null hypothesis by means of statistical tests. Since data is not normally distributed, for these tests we select the Mann-Whitney test (Sheskin(2007)) that estimates the median. To reject the null hypotheses we adopt the standard significance level at 5%, that is the probability of rejecting a null hypothesis when it is true (type I error).

Furthermore, to increase results reliability, we perform a sensitivity analysis and a validation of results. The sensitivity analysis is carried out by computing threshold ranges in which the composition of good/bad predictors sets remains the same: in this way we understand the impact of the thresholds choice on results, and we also exanimate border values. The validation is based on the idea that the good predictors effectively identify real bugs in the programs, therefore affecting their external quality, whereas the bad predictors are not related to defects and do not have impact on external quality. Hence quality of projects that contain good predictor issues detections should be lower than the mean quality of all the other projects, whilst quality of projects that contain bad predictor issues detections should be not different from the mean quality of the remaining projects. The proxy for projects' external quality is the percentage of passed tests in lab versions, positively correlated to the quality. Therefore we carry out the validation by comparing the proportion of acceptance tests passed by projects containing at least one occurrence of the issues in the set to be validated vs. the same proportion in the remaining programs.

3.1. Data Collection

An issue produced by FindBugs is characterized by an ID, a textual explanation, and a location in the source code. The issues are grouped by FindBugs in category (Bad Practice, Correctness, Style, Performance, and Malicious Code have at least one occurrence in the code base) and priority (Low, Medium, and High): hence the single issue is uniquely identified by the combination of ID, category, and priority. We store also their locations in the source code (file name, class, method, line number) and in the project (course ID, student ID, lab/home version). Afterwards, we use the DiffJ tool[1] to collect the changes done to evolve the lab version into the home version: DiffJ operates on two versions of a Java program and is able to compute for each pair of corresponding Java classes which lines changed. Finally, results of functional tests are obtained through the PoliGrader tool. The data collection process is represented in Figure 5.

3.2. Threats to Validity

We can identify three main threats: two external and one construct threat. The first external threat is: we study small student projects, hence the application of findings in industrial context is debatable. However, this weakness is balanced by the fact that this study eliminates the effect of developer style on the results, because a large pool of developers is used for the same projects. In addition, we recall the study of P. Runeson (Runeson(2003)), whose conclusions could neither reject nor accept the hypothesis on differences between freshmen, graduate students and industry people. We also draw in section 5.2 similar conclusions. The construct threat is concerning the identification of defects. We do not have a bug database but only tests failures: we make the assumption that all changes are done to fix a defect. It could be possible that some changes are not related to real defects, but to other

[1] available at http://www.incava.org/projects/java/diffj/

motivations (cleaner code, more readable code, and so on).

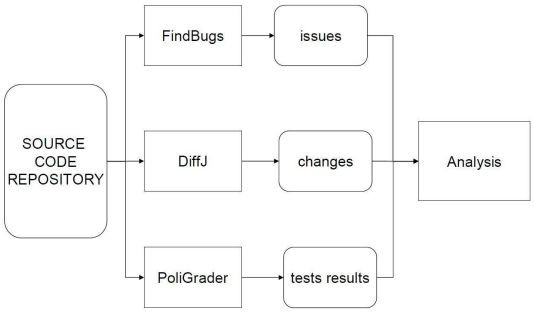

Figure 5. Data Collection Process

Nevertheless, we do not think that this approximation could affect change results, because students correct the lab versions doing as few changes as possible, for two reasons. Firstly, the home version is the last version of the project and actually no maintenance has to be done. Secondly students are discouraged from doing many changes because the mark suggested by PoliGrader decreases with the quantity of changes made (see details in (Torchiano and Morisio(2009))). However, the drawback is that many issues related to other aspects of quality beyond the correctness (for example maintainability or efficiency) could remain in the code and indicated as bad defects predictors, whilst in other contexts they could be fixed: this is the final thread.

4. Results

The automatic application of FindBugs on all the 301 projects (both versions, lab and home) produced a large collection of detections: 1692 in lab versions, belonging to 77 issues, whilst home versions detections are 1662, belonging to 73 issues (this does not mean that 30 issues were removed across all projects, since the number of issues in home version is given by: issues in lab version – issues fixed + new issues introduced). We answer to RQ1 computing Metric M1, that is the precision of the issues, with respect to temporal + spatial coincidence. Table 2 indicates minimum, maximum, 1st and 3rd quartile, median and mean of precisions (NA/NI) in projects.

The mean of precisions in projects is low (0.126) and the variability is high (standard deviation is 0.22, almost the double of the mean). More than 2/3 of projects have a precision lower than the selected minimum threshold 0.50 (only 6 projects out of 301 have a higher precision), and in half of the projects precision is about 1/5 of this threshold. These observations show that the threshold selected is very strict, despite of the initial considerations. Table 3 and Table 4 show the issues for which we could reject either of the two null hypotheses. We do not provide the precision of issues for which we can not reject either of the two null hypotheses because of their large number (77), however the full list is available on line[2].

The columns in the tables show: the issues ID, the average precision (sum of NA/sum of NI), the estimated median of precision, and finally the p-value of the Mann-Whitney single-tailed test.

Table 2. Distribution of issues precision

Min	1st q	Median	Mean	3rd q	Max
0	0	0	0.126	0.20	1

Table 3. Precision of good defect predictor issues

Issue ID	NA/NI	Prec. Est.	p-val
GC_UNRELATED_TYPES (Correctness,1)	12/15	1	0.048
SA_FIELD_SELF_ASSIGNMENT(Correctness,1)	7/10	1	0.012
UR_UNINIT_READ (Correctness,1)	6/7	1	0.012
UUF_UNUSED_FIELD (Performance,2)	26/55	0.5	0.045

Table 4. Precision of bad defect predictor issues

Issue ID	NA/NI	Prec. Est.	p-val
DM_NUMBER_CTOR (Performance,2)	0/6	0	0.018
DM_STRING_CTOR (Performance,2)	0/29	0	<0.01
DM_STRING_TOSTRING (Performance,3)	0/5	0	0.018
EQ_COMPARETO_USE_OBJECT_EQUALS (Bad_Practice,2)	5/275	0	<0.01
ES_COMPARING_STRINGS_WITH_EQ (Bad_Practice,2)	0/10	0	<0.01
IL_INFINITE_LOOP (Correctness,1)	0/5	0	0.036
NM_CLASS_NAMING_CONVENTION (Bad_Practice,2)	0/17	0	<0.01
NM_CONFUSING (Bad_Practice,3)	0/6	0	0.01

[2] http://softeng.polito.it/vetro/confs/ease2011/data.zip

NM_METHOD_NAMING_CONVENTION (Bad_Practice,2)	2/44	0	<0.01
NP_NULL_ON_SOME_PATH (Correctness,2)	0/4	0	0.036
OS_OPEN_STREAM (Bad_Practice,2)	0/71	0	<0.01
OS_OPEN_STREAM_EXCEPTION_PATH (Bad_Practice,3)	0/5	0	0.018
SE_BAD_FIELD (Bad_Practice,3)	0/11	0	<0.01
SE_COMPARATOR_SHOULD_BE_SERIALIZABLE (Bad_Practice,2)	0/49	0	<0.01
SIC_INNER_SHOULD_BE_STATIC_ANON (Performance,3)	0/92	0	<0.01
URF_UNREAD_FIELD (Performance,2)	33/259	0	<0.01

The set of good defects predictors is composed of 4 elements: 3 out of 4 have an estimated median precision of 1, the double that of the threshold. The median of the last issue, UUF_UNUSED_FIELD (Performance, 2), is exactly the threshold value: this is a border value and it will be examined in depth in Section 5. The 4 issues are:

- GC_UNRELATED_TYPES: a call to a generic collection method that contains an argument with an incompatible class from the collection's parameter.
- SA_FIELD_SELF_ASSIGNMENT: a self-assignment of a field, like `int y = y`.
- UR_UNINIT_READ: the constructor reads a field which has not yet been assigned a value.
- UUF_UNUSED_FIELD: a field is never used.

In contrast there are many more issues among the defect predictors set i.e. 16. All of them have median = 0. Since they are many, for their descriptions please refer to FindBugs website[3].

We perform a sensitivity analysis of results to check their stability with respect to the inclusion criteria: we compute the threshold ranges in which the composition of groups remains the same. The good predictors set is stable in the range 0.21–0.50. For threshold values greater than 0.5 the issues GC_UNRELATED_TYPES (Correctness,1) and UUF_UNUSED_FIELD (Performance,2) are excluded, and above 0.51 the set becomes empty. Analyzing instead lower bound, a new issue could be included in the set of good predictors only putting a very low threshold: at 0.20 issue NP_UNWRITTEN_FIELD (Correctness,2) could enter the group, and 2 more issues can enter with even lower thresholds : 0.12 and 0.11. Since the upper bound is already very strict and lower bound must be relaxed from 0.50 to 0.20 to change the set, we can affirm that results about good predictors are reliable.

The sensitivity analysis of bad predictors have the following result: the set is stable in the threshold range 0 – 0.15, so again a wide range. In fact we should use a high threshold, 0.16 (3 times bigger than the 5% of the original one) to change the set and include a new issue, NM_FIELD_NAMING_CONVENTION(BadPractice, 3). A further issue, REC_CATCH_EXCEPTION (Style,3), enters only with threshold = 0.25. We conclude that also bad predictors set is robust.

4.1. Validation of Good Defects Predictor issues

Figure 6 shows the boxplots of passed tests percentages in lab versions: NO_GP is the set of projects that do not contain any detection of a good predictor issue (259 projects), while GP is the set of projects containing at least one good predictor issue detection (the remaining 42 projects). The box plots clearly show that external quality of GP set is lower than external quality of NO_GP set: medians are respectively 63.64% and 40.91% . According to Mann-Whitney tests, we observe significant (p=0.001) differences between the two groups of projects. The 95% confidence interval for the difference between the medians is [6.29,∞]. There is strong statistical evidence that average external quality of projects with at least one good defect predictor issue detected is lower than the average externally quality of all projects. It is very likely that defects in projects with lower quality are correctly identified by the good predictor issues.

We continue the validation and we inspect all the good predictors issues signaled on source code, manually determining whether they were correctly detected by the tool and whether the problem signaled actually caused a wrong behavior of the program (failure of functional test). The detections of issues identified as good defects predictors are 87. We present the results of the manual code inspection in Table 5: for each issue, we indicate the ID, the total number of detections, the number of correct detections and the number of detections that impacted the functionality. Three issues of four have all detections correct and are the cause of an incorrect behavior of the program.

[3] http://findbugs.sourceforge.net/bugDescriptions.html

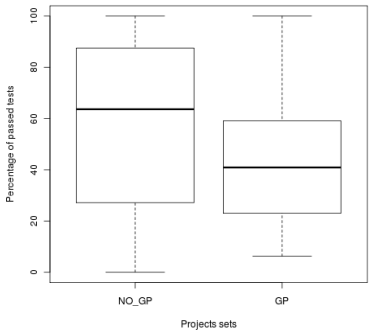

Figure 6. Box plots of passed tests percentages: good defect predictors

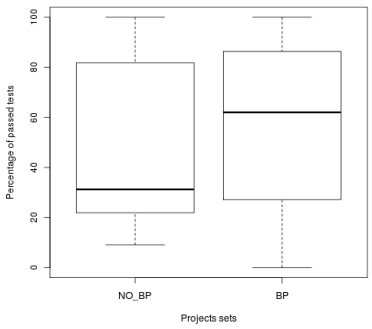

Figure 7. Box plots of passed tests percentages: bad defect predictors

Table 5. Manual inspection of good defects predictors issues

Issue	Nr of detections	Correct detections	Impact on functionality
GC_UNRELATED_TYPES (Correctness,1)	15	15	15
SA_FIELD_SELF_ASSIGNMENT(Correctness,1)	10	10	10
UR_UNINIT_READ (Correctness,1)	7	7	7
UUF_UNUSED_FIELD (Performance,2)	55	46	0

UUF_UNUSED_FIELD (Performance,2) is the exception: we discuss it in Section 5.

4.2. Validation of Bad Defects Predictor Issues

Figure 7 contains boxplots of passed tests percentages: NO_BP is the set of projects that do not contain any detection of a bad predictor issue (they are just 9), on the right BP is the set of projects with at least one detection of a bad predictor issue (292 projects). We observe that projects BP have higher percentages of passed tests then NO_BP projects. The medians are respectively 62.02% and 31.25%. However, the number of projects in NO_BP is so small that they cannot be a representative sample. In fact, although medians are so different, the null hypothesis that the two medians are equals cannot be rejected with α=0.05 and p-value is 0.1041 (according to Mann-Whitney test). The 95% confidence interval for the difference is $[-\infty,+3.75]$. We can therefore assume that no difference exists among the two sets. We do not perform a manual validation because of the high number of detections related to bad predictors issues (888 in lab versions): we consider the manual check of a representative sample of this bigger population an error prone task.

5. Discussions

5.1. Discussion on Results

Answer to RQ1. On the basis of the temporal + spatial coincidence criterion, issues related to defects are: call to unrelated types, field self assignment, uninitialized read of field in constructor (Correctness, 1), and unused field (Performance,2).

The manual validation (see Table 5) of detections showed that Correctness detections are all valid and have an impact on functionality. However, the Performance issue is the only one that has no correct detections that cause an incorrect behavior of the program. This fact is reasonable because the issue, as the name of the category suggests, is just signaling waste of memory (variable never used), and it is not a real error (because in this kind of little Java projects, performance of the program is neither mission nor safety critical). However, since their detections are about the 63% of all detections in the set (see Table 3), their contribution to the external quality prediction is important. In fact, there is a reason why projects with detections of unused fields have lower quality: their presence in a program means that the student encountered difficulties in the design of the program, because he planned to use more/different variables that in fact were not necessary. In contrast students who developed applications with higher external quality did not have this kind of problem. This is the reason why we decided to leave this issue in the set of good defect predictors issues, despite the category it belongs is Performance. Furthermore, the double validation process confirmed that all the 4 issues have a clear impact on external code quality and they can be considered as good defects predictors, with high confidence.

We also identify 16 issues that are bad defects predictors, and the statistical validation performed in Section 4.2 confirms that their detection has no correlation with the external quality of the projects.

However, we should be cautious with the bad defects predictors set, because the effect of the third threat to validity could affects this result. In fact, students must make as few changes as possible, otherwise their mark will decrease: for this reason, they just correct errors and do not perform any change related to performance, maintainability or even errors that are in impossible paths. Therefore, it is probable that in industrial projects some of these issues could be fixed by developers. Observing the type of issues in the set, we could assert that the majority of them could be related to this fact. For instance, 3 issues are naming convention violations, whose importance for code comprehension is well known, and 4 of the 5 issues belonging to the category Performance are memory leaks (useless constructor of String and Number, unread field and field that should be static). The fifth issue of Performance, i.e. useless `toString()` applied on a String, could indicate that students have not fully understood the nature of the objects in Java, as the GC_UNRELATED_TYPES "good" issue demonstrates. Also the issues on the comparison of Strings or Objects with `==` (Bad Practice) could be related to this problem. The remaining issues of category Bad Practice do not signal bugs but do indicate code that could lead to a waste of resources or to difficulties in maintenance. Finally, there are 2 issues in the category Correctness in the list: the infinite loop and the null pointer dereference in some path. We checked them manually and we discovered that they are actually errors: however all the 9 detections are on unfeasible paths, and this is probably the reason that students did not notice these errors with tests execution. Thus, we decided to remove the two issues of category Correctness from the list of bad predictors.

An important practical application of the findings is a filtering strategy that can avoid information overload on developers caused by a very large number of detections. In particular fixing issues with a low probability of being related to a defect is dangerous because we know from Adam's law (Adams(1984)) that the probability of introducing a new error during a fault correction is always greater than zero. The ranking could be adopted by developers that want to enable only those issues with the highest precision. For instance, in this experiment, the good defects predictors issues are just 4 out of 359 in FindBugs 1.3.8 database (about 1%) and they are responsible for only the 4.4% of all detections in lab versions. Instead, the bad predictors issues (about 4% of the complete set) produced about the 45% of detections in lab versions.

Furthermore, from an educational perspective, although the occurrences of good predictors are few, we consider them important topics to be stressed more in future iterations of the OOP course.

5.2. Comparison with Previous and Related Work

In our previous work (Vetro' et al.(2010)Vetro', Torchiano, and Morisio), we analyzed a smaller repository of OOP projects (85 projects) and we studied the precision of issues at group level (combination of category and priority): the analysis demonstrated that only 2 groups (*Bad Practice High*, *Correctness High*) out of 15 groups of issues could be considered as reliable predictors of actual defects, and one group of issues (*Bad Practice Medium*) had a precision that was practically negligible.

We group the good and bad defects predictors issues found in this study with the same criteria of the previous work to compare the findings. Since the repository in the replicated study is bigger, we find more issues and more groups, however the group *Correctness High* is still in the good defects predictors set as *Bad Practice Medium* is still the Bad Defect Predictors set. The group *Bad Practice High* instead is not present in either of the two sets. Therefore, 2 out of 3 groups are confirmed in the respective sets and there are no conflicts in the group compositions of the two studies: we conclude that the finding of our previous work are confirmed and improved in this study.

Before us, Boogerd and Moonen (Boogerd and Moonen(2008)) (Boogerd and Moonen(2009)) and Kim and Ernst (Kim and Ernst(2007)) also used temporal and spatial coincidence in their research. Our research confirms the findings of Boogerd and Moonen: a reduced set of rule violations are related to defects in source code, and many violations are not related to real defects. Furthermore, in our analysis there are 3 high priorities issues and 1 medium priority in the good defects predictors, whilst the majority of issues in the bad predictors set are medium and low priority (respectively 10 and 5 issues, 1 high priority): thus the tool's default prioritization of issues seems to be effective, in contrast with what is found by Kim and Ernst (Kim and Ernst(2007)) in open source projects. In the same study, the authors list the FindBugs issues with shortest and longest "life" in multiple versions of two open source projects: the underlying idea is that if some issues are resolved quickly by developers, those issues are important and likely related to real defects. The issue self assignment of field, that we identify as good defect indicator, is also among the issues with shortest life in one of the two projects analyzed by Kim and Ernst. The bad defect predictors issues in common are instead two: ES_COMPARING_STRINGS_WITH_EQ and

OS_OPEN_STREAM, both of (BadPractice,2). Furthermore, we do not observe any conflict, i.e. none of our "good" issues have long life in the experiment of Kim and Ernst and none of the "bad" have short life.

We are also able to compare our findings with the findings of Ayewah and Pugh (Ayewah and Pugh(2010)), who analyzed thousands of FindBugs warnings fixed by engineers during the May 2009 "Google FixIt". The authors used a lightly modified temporal coincidence to find which FindBugs issues were fixed with higher frequency in the Google code base in a period of 9 months. In their paper they show 12 issues with high removal rate and 3 with low removal rate, distinguishing issues only by bug pattern and category. We found in the first set 3 out of 4 of our "good" issues (only the self assignment is not present). Moreover, "our" bad issue DM_NUMBER_CTOR (Performance) has low fix rate. The only conflict found is provoked by the issue NP_NULL_ON_SOME_PATH, that was originally in our bad predictors set and has a high removal rate in the other study: this confirms our choice to take it out from the set of bad defects predictors (see 5.1).

In a previous work of the same authors, (Ayewah et al.(2007)Ayewah, Pugh, Morgenthaler, Penix, and Zhou), they tried to understand the efficiency of FindBugs by manually checking medium/high priority Correctness issues signaled on 3 projects. The authors further classified issues in 4 groups, based on their impact on code. Overall, they observe that in JDK 1.6.0-b105 almost 50% of them had an impact (misbehavior of the program), a further 10% had a serious impact, 160/379 were trivial, whilst 5 issues were due to bad analysis by FindBugs. We find among the issues with at least one impact or serious impact the uninitialized read of field in constructor (1 detection had impact and 7 tagged as trivial) and the self assignment of field (1 "impact" and 2 "trivial"). Thus, there is no conflict between our study and theirs. In the same paper the authors provide the results of a similar review that was performed at Google, where they classified issues reviewed in impossible (i.e. wrong detection), trivial, open, fixed. Looking at the results, 2 out of 13 uninitialized reads were fixed (but 7 were wrong), whilst all the 7 detections of the issue GC_UNRELATED_TYPES (Correctness) were still open. Among the 12 detections of the self assignment issue, 5 were fixed, 1 classified as trivial and the remaining left open. Among the set of our bad issues we found two conflicts, because both the two Correctness issues in the initial set of bad defects predictors were instead related to real defects in the Google Code Base. In fact, 30 out of 31 warnings of infinite loop were corrected and 35 out of 98 detections of possible null pointer deference were fixed (but 10 tagged as trivial and the remaining half was a detection error or still open). However, since after discussion in 5.1 we removed these issues from the set, this conflict also confirms our decision.

In summary, the bad defects predictor issues in common with other studies are few: the low reliability of bad issues set is explained by the construct threat, because students fixed only code that caused a test failure, and did not look to other aspects like performance and maintainability that are signaled by FindBugs issues. This fact, on the flip side, together with the double empirical validation that we conducted, make results on the "good" issues very reliable, because we have a high confidence that changes were made to fix errors, and the disappearing issues were related to that errors. Additionally, we observe that these issues are a subset of those ones identified by other studies conducted in industrial and open source projects. Therefore, we can assert that the empirical evidence of the goodness of these issues is growing in literature. For this reason, these issues could have higher priority than others and ease the tool customization, having a practical impact of filtering issue notifications for developers that should reduce the information overload. Furthermore, the adoption of our modification of the temporal + spatial technique with information on test failures, could be used in other contexts. In fact, issues with highest precision can be identified in programs that are already tested and then used to check software that is still in production code: in this way many bugs could be found before the testing phase, when the removal cost is lower.

Finally, we can also assert that our main external threat (study on small students projects) has a weak impact, because our results are generally consistent with the findings of similar studies.

6. Conclusions and Future Work

We analyzed the relationship between FindBugs issues and defects on 301 University Java projects, using information on changes in source code and tests failures. We obtained that only 4 issues could be considered as reliable predictors of real defects and 14 issues had a negligible precision. We compared the results with our previous work, confirming the former findings. Subsequently we compared our results with three similar studies in the literature: we found few intersections for the set of bad defects predictor issues. However, the issues we classified as good defect predictors were also identified as related to defects by other researchers, and no conflicts were found. In

summary, the main contributions of this work are: we provide more empirical evidence about the validity of some issues as bug predictors (I) and we improve the temporal + spatial coincidence technique using tests failures information (II).

Our future work will be devoted to a repetition of this study on industrial and open source projects.

7. References

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

[Ayewah and Pugh(2010)] Ayewah, N., Pugh, W., 2010. The google findbugs fixit. In: Proceedings of the 19th international symposium on Software testing and analysis. ISSTA '10. ACM, New York, NY, USA, pp. 241–252.

[Ayewah et al.(2007)Ayewah, Pugh, Morgenthaler, Penix, and Zhou] Ayewah, N., Pugh, W., Morgenthaler, J. D., Penix, J., Zhou, Y., 2007. 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. ACM, New York, NY, USA, pp. 1–8.

[Basili et al.(1994)Basili, Caldiera, and Rombach] Basili, V., Caldiera, G., Rombach, D. H., 1994. The goal question metric approach. In: Marciniak, J. (Ed.), Encyclopedia of Software Engineering. Wiley.

[Boehm and Basili(2001)] Boehm, B., Basili, V. R., 2001. Software defect reduction top 10 list. Computer 34 (1), 135–137.

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

[Boogerd and Moonen(2009)] Boogerd, C., Moonen, L., 2009. Evaluating the relation between coding standard violations and faults within and across software versions. In: MSR '09: Proceedings of the 2009 6th IEEE International Working Conference on Mining Software Repositories. IEEE Computer Society, Washington, DC, USA, pp. 41–50.

[Hovemeyer and Pugh(2004)] Hovemeyer, D., Pugh, W., 2004. Finding bugs is easy. In: OOPSLA '04: Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications. ACM, New York, NY, USA, pp. 132–136.

[Kim and Ernst(2007)] Kim, S., Ernst, M. D., 2007. Prioritizing warning categories by analyzing software history. In: MSR '07: Proceedings of the Fourth International Workshop on Mining Software Repositories. IEEE Computer Society, Washington, DC, USA, p. 27.

[Li et al.(2006)Li, Tan, Wang, Lu, Zhou, and Zhai] Li, Z., Tan, L., Wang, X., Lu, S., Zhou, Y., Zhai, C., October 2006. 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.

[Runeson(2003)] Runeson, P., 2003. Using students as experimental subjects - an analysis of graduate and freshmen student data. In: Proceedings of the 7th International Conference on Empirical Assessment in Software Engineering (EASE 2003). pp. 95–102.

[Sheskin(2007)] Sheskin, D. J., January 2007. Handbook of Parametric and Nonparametric Statistical Procedures, Fourth Edition. Chapman & Hall/CRC.

[Torchiano and Morisio(2009)] Torchiano, M., Morisio, M., 2009. A fully automatic approach to the assessment of programming assignments. INTERNATIONAL JOURNAL OF ENGINEERING EDUCATION 24 (4) (0), 814–829.

[Vetro' et al.(2010)Vetro', Torchiano, and Morisio] Vetro', A., Torchiano, M., Morisio, M., 2010. Assessing the precision of FindBugs by mining java projects developed at a university. In: MSR. pp. 110–113.

[Wagner et al.(2008)Wagner, Deissenboeck, Aichner, Wimmer, and Schwalb] Wagner, S., Deissenboeck, F., Aichner, M., Wimmer, J., Schwalb, M., 2008. An evaluation of two bug pattern tools for java. In: ICST '08: Proceedings of the 2008 International Conference on Software Testing, Verification, and Validation. IEEE Computer Society, Washington, DC, USA, pp. 248–257.

[Wagner et al.(2005)Wagner, Jürjens, Koller, Trischberger, and München] Wagner, S., Jürjens, J., Koller, C., Trischberger, P., München, T. U., 2005. Comparing bug finding tools with reviews and tests, 40–55.

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